Trabajo de Fin de Grado

Doble Grado en Ingeniería Eléctrica y Electrónica

Optimización de la capacidad y ciclo de carga y descarga de una batería para reducir el coste eléctrico en función de la predicción de demanda y del precio de la energía.

Autor: Javier Adán berlanga Linero

Tutor: Félix Biscarri Triviño

Departamento de Ingeniería Electrónica

Escuela Politécnica Superior

Universidad de Sevilla

Sevilla, 2025

Trabajo de Fin de Grado

Doble Grado en Ingeniería Eléctrica y Electrónica

Optimización de la capacidad y ciclo de carga y descarga de una batería para reducir el coste eléctrico en función de la predicción de demanda y del precio de la energía.

Autor: Javier Adán berlanga Linero

Tutor: Félix Biscarri Triviño

PALABRAS CLAVE

Ahorro económico, Optimización energética, Batería eléctrica doméstica, Gestión de batería, Python, Inteligencia Artificial, Predicción de demanda.

ABSTRACTO

Con el objetivo de ahorrar en la factura eléctrica, se usará una batería controlada activamente para cargarla en las horas con el precio de la electricidad más bajo, y descargarla cuando se dé una demanda en las horas punta, en las que será más cara. Se ha desarrollado un *software* que calcula los ciclos y capacidad de batería óptimos para la demanda y el caso concreto del usuario. Luego con esa batería instalada, el mismo *software* usará una IA personalizada, entrenada en los datos específicos del usuario, para predecir sus demandas y precios. Con ellos calcula cada día el ciclo óptimo que debe ejecutar la batería el próximo día.

AGRADECIMIENTOS

A mis padres y familia, que me apoyaron y me permitieron llegar hasta aquí, y me ayudaron a superar épocas de dificultades y dudas.

A mis amigos, tanto los que conocí en persona como los que conocí en línea, que me permitieron desarrollar mis gustos y aficiones, y mejorar como persona.

Javier Adán Berlanga Linero

Sevilla, 2025

ÍNDICE

ÍNDICE DE FIGURAS

RESUMEN

Con el objetivo de ahorrar en la factura eléctrica, se va desarrollar un método alternativo y complementario a los clásicos de ahorro, siendo los métodos clásicos la instalación de renovables para autogeneración, y usar equipos y sistemas más eficientes energéticamente (reducir la demanda eléctrica).

Este método alternativo está basado en la fluctuación natural del precio horario de la electricidad a consecuencia de la ley de oferta y demanda. Es decir, a las horas que más se produce y/o menos se consume el precio de la electricidad es más bajo, de modo que realizar los grandes consumos eléctricos durante este periodo resulta que el cómputo total de la factura sea más barato, generando un ahorro. La forma más inmediata de hacerlo es simplemente cambiar los hábitos de consumo y adaptarse a estos precios fluctuantes, pero no siempre es posible ni deseable. Entra aquí el concepto que se desarrollará en este proyecto, la **instalación de una batería de uso doméstico** en la propia casa. Una batería no demasiado grande ni cara, que sea capaz de **cargarse en las horas más baratas y descargarse en las más caras**, aprovechando estas fluctuaciones de precio horario sin la necesidad de cambiar hábitos de consumo.

El problema a resolver será predecir las demandas del usuario, así como los precios de la electricidad para el próximo día. Con estos datos, y teniendo una batería instalada se podrá **optimizar el ciclo** de esta para buscar el precio mínimo (nuevamente, cargar en horas baratas, descargar cuando haya demanda en horas caras, pero usando una solución matemáticamente optimizada).

Aparecen dos cuestiones en el proceso. La primera, ¿cómo hacer la predicción de demandas y precios? Se usará una **inteligencia artificial** entrenada con los datos específicos y particulares del usuario, de modo que sea capaz de predecir el ciclo próximo particular del usuario. Análogo para los precios, aunque esta vez no serán particulares del usuario, pero serán datos para todo el país. Se deberá no solo crear el modelo de inteligencia artificial, pero también entrenarlo y ejecutarlo diariamente para que el sistema funcione correctamente durante la vida útil de la batería.

La segunda cuestión es, ¿qué capacidad de batería se ha de instalar? Las baterías no son gratis, una muy grande ocasionará que el precio de la inversión en esta superará el ahorro que se pueda obtener del sistema. Una batería muy pequeña y el sistema será poco efectivo, y el ahorro conseguido de este será bajo en consecuencia. Aparece no solo la optimización previa del ciclo, también la optimización de la capacidad de la batería en sí misma, en función del precio que se pueda ahorrar y del precio que se podrá invertir. Se tomarán los datos particulares de demandas del usuario y se hará un complejo cálculo de

optimización que indique qué **capacidad de batería se habrá de instalar en función del precio de mercad**o. Un mercado con baterías baratas favorecerá la instalación de baterías más grandes, haciendo el sistema más efectivo con una inversión no tan alta y permitiendo más ahorro (siempre que el perfil de demandas del usuario permita este ahorro). Caso contrario si el mercado no es favorable, siendo el peor caso que no merezca la pena la instalación del sistema por un coste inicial demasiado elevado respecto al ahorro esperado durante su vida útil.

Se desarrollará e implementará un ***software* escrito en *Python*** que tome estos datos de demanda del usuario, así como todos los datos necesarios de internet, y que sea capaz de procesarlos autónomamente (con la mínima intervención posible del usuario). Responderá así ambas cuestiones, tanto la capacidad óptima en función de las demandas del usuario y el precio de mercado de las baterías, así como si se decide instalar esta batería, el ciclo óptimo de esta cada día (ejecución del programa recursivamente todos los días, usando la IA personalizada para predecir los datos del próximo día).

DATOS Y HERRAMIENTAS

Se van a exponer las herramientas usadas y mencionadas, incluyendo los repositorios al código fuente del *software* referenciado y desarrollado en este documento.

- Repositorio principal del *software*, **GitHub** (GitHub): https://github.com/jadanberlanga/TFE-Control-y-optimizacion-de-bateria

- Repositorio secundario del *software*, **Google Drive** (Google): https://drive.google.com/drive/folders/1pwBFpQaYA0V5KJdP3U_gZG_HncByipeW?usp=sharing

- El código ha sido escrito usando el editor PyCharm (Community Edition) (Jetbrains).

- Los gráficos se han hecho con AutoCad (Autodesk).

INTRODUCCIÓN

La gestión de la energía es uno de los aspectos claves que enfrenta un país en su progreso y avance. Influye en todo tipo de aspectos, desde capacidad de producción, estabilidad, riqueza, así como impacto sobre el medio ambiente. Es tan importante que es un índice de avance de un país (International Energy Agency, 2024).

Se destina una gran cantidad de recursos tanto económicos como humanos para desarrollar una buena red eléctrica y energética, con una generación de energía sostenible y estable. En España concretamente se ha hecho una fuerte inversión en las renovables (Red Eléctrica de España (REE), 2025), especialmente en la energía solar debido a la ubicación geográfica favorable que tiene, con gran cantidad de horas solares, y espacio disponible para su instalación. Sin embargo, está lejos de ser un país que solo pueda abastecerse de renovables, aún sigue teniendo una gran dependencia de otras fuentes no renovables como el gas natural (International Energy Agency, 2024).

Sin embargo, España no tiene grandes reservas de gas natural así que está forzada a importarlo, y en periodos de inestabilidad política esta importación, así como otros factores, viene asociada con un incremento monetario sustancial (MEDREG (Mediterranean Energy Regulators), 2023), el cual repercute en todos los aspectos a los que afecta la gestión energética dentro del país. Entre ellos al consumidor final, que verá su factura energética multiplicada por este aumento de la energía. Es por tanto de interés particular del consumidor final buscar **formas de reducir la factura eléctrica**.

La factura eléctrica depende de muchos factores, la mayoría fuera de control directo e inmediato del usuario, tales como impuestos y tasas o el precio horario de la electricidad. El parámetro que más puede controlar el usuario es la demanda eléctrica. Es decir, el consumo. La forma más inmediata de bajar el precio de la factura eléctrica es por tanto simplemente consumir menos. Esto se puede conseguir haciendo sacrificios (por ejemplo, usando una climatización más suave, calentar menos en invierno y enfriar menos en verano) o instalando equipos más eficientes (siguiendo el ejemplo, mejorar el aislamiento térmico de la vivienda e instalar un sistema de climatización más eficiente energéticamente).

Al ser la forma más directa esta opción es la forma más efectiva de reducir el coste de la factura, y debería aplicarse siempre que sea posible. Pero no siempre es posible o no siempre es suficiente. Se deben buscar entonces otros enfoques de ahorro. Por ejemplo, generar parte del consumo eléctrico, autogeneración. Como se ha mencionado, España tiene el privilegio de tener abundancia solar, es factible y común instalar paneles solares para generar parte de la demanda

consumida, siendo uno de los líderes mundiales en este nicho (IEA PVPS, 2023 (informe, versión 2024/2025)). Pero el principal problema de la solar es que solo genera cuando hay sol, y es habitual tener actividad y consumo fuera de estas horas solares. Se podría complementar esta energía con otro tipo de energía renovable, pero en general todas tienen un problema similar, no están siempre disponibles, o no en la cantidad deseada en el momento requerido. La solución tradicional a este problema es la instalación de baterías, capaces de almacenar energía en periodos de abundancia y descargarla cuando se necesite. Pero estas baterías eventualmente se van a descargar, es una solución parcial. Suficiente para la inmensa mayoría de los usuarios, pero parcial, al fin y al cabo.

Se debe plantear entonces una pregunta, ¿existen otras formas de ahorro energético que no pasen por consumir menos o generar parte de la demanda consumida? La respuesta es sí, y está basada en uno de los parámetros que en principio no son controlables por el usuario, el precio horario de la electricidad. Por el funcionamiento del mercado eléctrico en España, el precio de la electricidad varía durante el día (OMIE (Operador del Mercado Ibérico de Energía)). Si se tiene un contrato en el que el precio de la energía consumida se ajuste al de mercado hora a hora, es posible acceder a una forma de ahorro más. Esta es simplemente variar los hábitos de consumo, y centrar mayores gastos en las horas más baratas. De esta forma, aún si la demanda total es la misma, al haber cambiado el ciclo de demanda y ajustarlo a las horas más baratas, el importe por esta demanda será menor.

Sin embargo, esta última forma de ahorro implica una concesión en cuanto a cambiar los hábitos de consumo, los cuales no siempre será posible o simplemente no se deseará. ¿Existe por tanto una forma de **bajar el precio de la factura eléctrica sin consumir menos y sin cambiar los hábitos de consumo**? Esta es la pregunta que se responderá y tratará en este proyecto.

HIPÓTESIS

Se parte de la pregunta planteada en la introducción, ¿existe una forma de bajar el precio de la factura eléctrica sin consumir menos y sin cambiar los hábitos de consumo?

La primera clave para responder esta pregunta es que el precio horario de la electricidad varía durante el día. Se puede consultar la página de la entidad encargada de esta tarea en España, OMIE, para ver este ciclo. Entonces mientras se tenga un contrato que refleje estos cambios de precio, **el consumo**

eléctrico en unas horas será más barato que el consumo en otras, bajando el total del precio de la factura eléctrica.

Pero este cambio de consumo tiene un cambio de los hábitos intrínseco, se han de revisar estos precios diariamente para identificar las horas más baratas, y cambiar activamente los patrones diarios para adaptarse a estos precios que fluctúan diariamente. Entra aquí un elemento capaz de almacenar energía y **desplazar el consumo a una hora deseada** para su posterior uso. Una batería.

Ambas técnicas, la de cambiar hábitos de consumo para adaptarse a las horas más baratas, así como la instalación de baterías (especialmente junto a la instalación de una fuente de energía renovable para autogeneración) para ahorro energético son técnicas clásicas y probadas, y comúnmente citadas como métodos efectivos y probados de ahorro (Manembu, y otros, 2023). Sin embargo, el uso de ambas técnicas en conjunto no es algo que se haya explorado. Y tiene un motivo claro, se necesita un sistema de control complejo, capaz de adaptarse a los precios horarios y a la demanda del usuario en tiempo real. Nace así la hipótesis en la que se basa este proyecto, la pregunta inicial se transforma en ¿es posible **instalar una batería y controlarla**, de forma que pueda **adaptarse a la demanda del usuario y al precio de la electricidad a tiempo real**?

Para poder comprobar esta hipótesis la primera cuestión que aparece es cómo se pueden obtener datos de demanda del usuario y precios de la electricidad a tiempo real. Y no solo a tiempo real, pues para poder adelantarse a las demandas del usuario para darle tiempo a la batería a cargar se necesita una predicción. Una **predicción de demandas** y de precios horarios. Esto por sí solo ya es una cuestión compleja en la que hay gran cantidad de estudios y modelos ya realizados, pero que en esencia sigue siendo por naturaleza un problema abierto.

La siguiente cuestión es que aún si se hace una buena predicción de demandas del usuario y de precios de la electricidad, ¿cómo se puede calcular el ciclo de demanda de la batería? Se puede intuir que se debe cargar en previsión a una demanda a una hora en la que el precio sea barato, balanceando energía consumida y cedida por la batería. Pero la intuición no es suficiente para este problema, se necesita un método matemático para resolver este problema. En esencia, se necesita un algoritmo de **optimización del ciclo de la batería**, que busque obtener el precio total más bajo posible, teniendo en consideración el total de demandas horarias del sistema (tanto de batería como del usuario, así como otras posibles fuentes) y el precio también horario de la energía. Ya se puede prever que este no será un problema trivial, así que se requerirán herramientas computacionales.

Pero una vez resueltas estas cuestiones, que si bien no son simples son factibles, parece a falta de cálculos que el sistema podría ser viable. Pero durante este planteamiento se ha obviado una cuestión clave, el precio de la batería. Las baterías no son gratis, y si se habla de ahorrar en la factura eléctrica usando una batería, entonces también se debe incluir la influencia del precio de esta en el cálculo. Se puede intuir también que una batería más grande será más eficaz en el propósito perseguido que una más pequeña, dando un mayor ahorro. Pero también una batería más grande tendrá un coste superior.

Además, diferentes perfiles de demanda tendrán más compatibilidad con este sistema y tendrán mayor o menor potencial de ahorro, se deberán estudiar casos particulares para determinar este potencial. Mientras mayor sea este ahorro posible, más grande podrá ser la inversión en la batería comprada, que a su vez permitirá sacar más ahorro del sistema. Análogo es el caso opuesto, una demanda pequeña o un perfil de esta no particularmente compatible con este sistema de desfase con la batería no podrá conseguir un ahorro que justifique una batería más grande, haciendo que a su vez esta menor batería sea menos efectiva.

Por tanto, partiendo de la suposición de que el sistema es funcional en teoría, la pregunta vuelve a mutar. La pregunta ya no es un simple sí se puede instalar una batería y controlarla para generar ahorros. La pregunta ahora es qué **capacidad es la óptima para maximizar este ahorro**. Una batería muy pequeña tendrá poco impacto, pero una mayor puede costar más la inversión inicial que lo que se podrá ahorrar en su vida útil. El problema pasa de ser uno puramente técnico, a tener también una parte económica asociada. Se deberá estudiar no solo la **viabilidad técnica**, sino también la **económica**. Y como todo problema económico va a haber un rango de valores en los que será viable económicamente, y otro que o bien simplemente no lo sea, o que el ahorro sea tan bajo que no merezca la pena su instalación y ejecución.

Y por la naturaleza del problema, esta pregunta no podría ser respondida genéricamente, pero en su lugar se deberán tomar los datos de un usuario concreto y hacer un análisis particular. No solo para calcular esta capacidad de batería óptima (si es que hay alguna que merezca la pena siquiera) y poder instalarla, sino luego controlarla durante toda su vida útil para poder obtener este ahorro previsto.

Además, no solo existirá la parte teórica, que calculará valores y ciclos óptimos, y la viabilidad general del problema. También deberá existir una parte física que pueda ejecutar estos ciclos calculados, pero debido a la naturaleza diferente y separada de este cálculo, en este proyecto no se abarcará la implementación física, considerándose un proyecto de envergadura similar al cálculo teórico.

MÉTODO

Planteamiento del problema

Se parte de la hipótesis planteada, buscar el rango óptimo en el que sea factible económica y técnicamente el que un usuario pueda instalar una batería no excesivamente grande y cara en su casa, de forma que pueda usar las fluctuaciones del precio de la electricidad hora a hora a su favor para poder abaratar el coste de la factura eléctrica. Y si se encuentran soluciones óptimas y el usuario decide instalar dicha batería, controlarla diariamente para ejecutar la solución prevista.

El problema como ya se ha teorizado será altamente **personalizado**, teniendo cada usuario una solución particular. Por tanto, el usuario deberá proporcionar sus datos únicos de demanda, mientras más precisos mejor. El resto de parámetros no son particulares, así que habrá métodos de obtenerlos que no pasen por requerir más información del usuario.

Con esto se tendrá una base de partida con idealmente una gran cantidad de datos sobre los que trabajar, a los que se podrán aplicar herramientas de procesado de datos modernas para obtener soluciones precisas, tanto para obtener el rango óptimo, como luego el control diario si el usuario decide instalar el sistema. Se puede plantear el proyecto entonces como un **problema a resolver**, uno de **optimización** en esencia.

El problema por tanto implica cálculos de optimización complejos con miles de datos. Esta condición ya elimina prácticamente todas las soluciones manuales. Esto deja casi exclusivamente como opciones viables soluciones que pasen por un ordenador. Además, la naturaleza del mismo implica que deberá resolverse una versión más reducida del mismo diariamente, por lo que eso también impone que la solución pueda ser automatizable, descartando programas y *softwares* tradicionales de cálculo también.

Esto significa que la naturaleza del problema hace que la inmensa mayoría de soluciones tradicionales no puedan ser usadas. Se deberá desarrollar entonces una solución personalizada. Esto es, escribir el código de un programa que resuelva este problema, un *software* personalizado.

Hay muchas opciones para desarrollar este programa, tanto en “plataformas” como en lenguajes de programación. Se elige ***Python*** (Python Software Foundation) por la cantidad de librerías de acceso libre ya existentes, así como

por su sencillez de uso respecto a otros lenguajes como podría ser *C* o *Java*. Cabe destacar que *Python* es conocido por ser menos eficiente que otros lenguajes, por lo que en principio no parece ser la mejor solución para un cálculo que va a trabajar con miles de datos. Sin embargo, esto no será un problema. Un buen código en *Python* no resolverá el problema significativamente más lento que otros lenguajes de programación. Simplemente se deberá hacer énfasis en desarrollar un **código eficiente** y optimizado. Las ventajas de *Python* siguen dándole la ventaja respecto a otros para esta tarea.

El problema tendrá **dos partes**. Una de "configuración", y otra de "ejecución diaria" (usar la configuración calculada). Se optará por un mismo *software* con dos modos de activación, uno para cada parte. De esta forma se podrán reusar funciones, y para el usuario será más sencillo tener una única aplicación que pueda "hacerlo todo".

La "configuración" pasa por calcular la batería óptima para el caso particular y personalizado del usuario, usando sus datos particulares. El *software* deberá analizar grandes cantidades de datos, calculando y optimizando usando estos datos para diferentes escenarios. Se deberá hacer un cálculo complejo, pero solo una vez. Esto le dará al usuario la información necesaria para que pueda hacer una elección educada para **elegir la batería óptima**.

Una vez obtenida esa batería ya se tendrá la configuración necesaria. Con eso se podrá resolver el problema diario, es decir, hacer una predicción de los datos eléctricos para que el *software* pueda prever el **ciclo óptimo de la batería para el día siguiente**. La batería se deberá adelantar a la demanda del usuario cargando en las horas en las que el precio de la electricidad es más barato, y descargando en las que la demanda del usuario y el precio de la electricidad son mayores.

Ya que se ha decidido usar un mismo *software* para ambos modos, el código deberá tener una forma sencilla de alternar entre estas dos opciones, de la forma más transparente y sencilla para el usuario. Es decir, una vez esté la configuración realizada, que cada modo se pueda activar con "un botón". Esto permitirá hasta a un usuario no técnico poder ejecutar el programa fácilmente. Aunque la configuración, si bien se deberá diseñar para que sea sencilla, puede requerir cierto conocimiento técnico.

Estructura del código

En este apartado se planteará la estructura general del código a un nivel muy alto de abstracción. Es objetivo de este apartado entender el flujo general del código. Si se necesitan más detalles de alguna parte se incluye al final de este documento un capítulo de documentación (así como el código fuente documentado y con comentarios, adjunto a esta memoria si se necesitan las soluciones específicas utilizadas).

Como ya se ha dicho el *software* tendrá **dos modos** de ejecución distintos, cada uno con su propia línea de ejecución y objetivo. Sin embargo, como también ya se ha dicho, ambos modos comparten una base común. Aparecen así tres partes diferenciadas en la estructura del código. La primera parte, la base común de ambos modos de ejecución, referenciada de ahora en adelante como "**Parte Común**". Esta parte tendrá como objetivo gestionar la entrada de datos y procesarlos. La segunda parte será la parte específica y necesaria para obtener la "configuración" previamente mencionada, la parte que calculará la capacidad de la batería que se va a comprar e instalar usando los datos históricos que se tienen. Se usará el nombre de "**Modo Histórico**" para referenciar a esta parte (debido al uso de los datos históricos). Su objetivo será tomar los datos obtenidos y para un rango de precios de baterías en el mercado, ejecutar un cálculo de optimización complejo para cada uno de los precios de este rango, obteniendo y recomendando la capacidad de batería óptima que se deberá comprar e instalar según los precios de mercado. Por último, la parte final será la que se ejecutará diariamente para predecir la solución a aplicar al día siguiente para optimizar costes. Deberá predecir datos futuros (para el día siguiente al menos, es decir el día objetivo) y optimizar el ciclo de demanda de la batería basada en estas predicciones. Se identificará a esta parte como "**Modo Futuro**" desde ahora. Su objetivo y función será obtener dicho resultado y exportarlos no en "formato humano", sino en un formato que otro controlador pueda entender fácilmente ya que será esta la información que tendrá la batería durante todo el día para funcionar.

El flujo general del código será primero elegir qué se quiere calcular y obtener, o bien la capacidad de batería óptima, o el ciclo de demanda de dicha batería para el día siguiente al cálculo. Es decir, qué modo del *software* se ejecutará, el Modo Histórico o el Modo Futuro. Esta decisión determinará que partes del código se ejecutarán (La elección del histórico implica que no es necesario ejecutar la parte de futuro, y viceversa). Pero independientemente del modo ejecutado, los datos de entrada serán los mismos, así como el código ejecutado en primer lugar, la Parte Común. Se puede visualizar esto con la Figura 1.

Una vez entendido el proceso, se deben desarrollar estas partes del código.

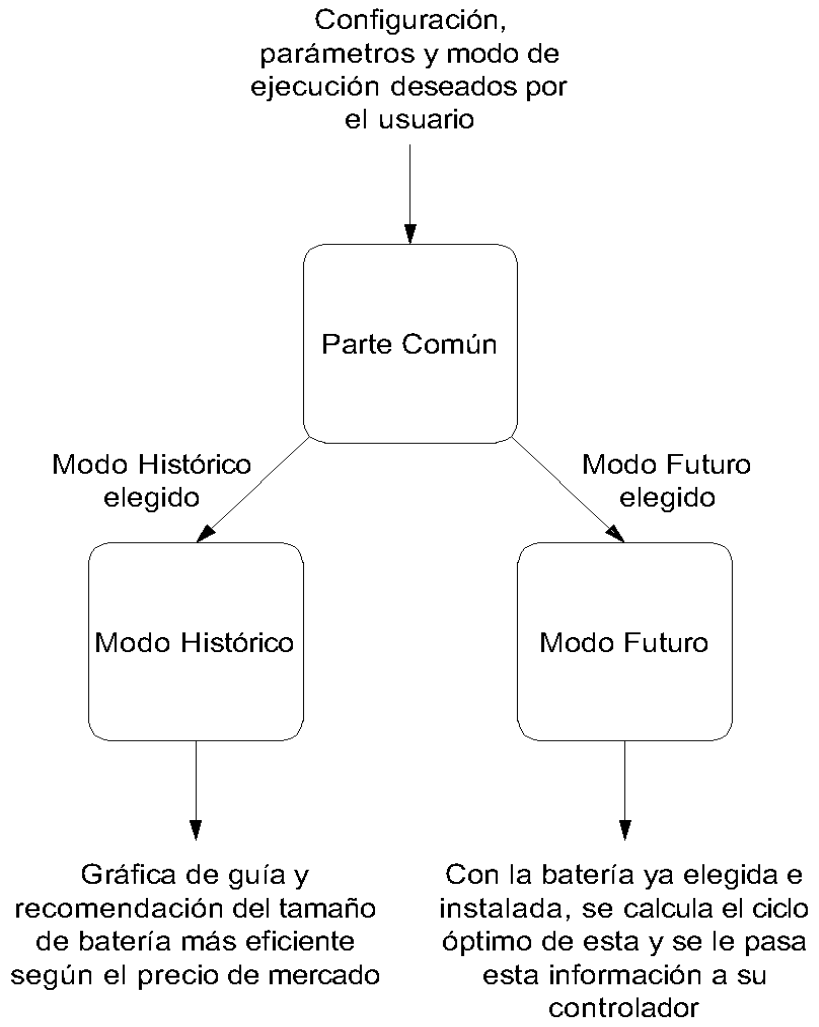


Figura 1. Estructura del código simplificada.

Parte Común

Esta será la primera parte que ejecute el código. Es la encargada de gestionar toda la información del problema, tanto información y peticiones del usuario, como la obtención de los datos necesarios para la ejecución del algoritmo de optimización que se usarán en la siguiente parte (ya sea la de históricos o la de futuro).

Lo primero que hace es leer la configuración. Es un archivo donde el usuario escribirá todos sus datos particulares y condiciones requeridas. En este archivo

se definirán datos como potencia máxima contratada, coordenadas de longitud y latitud, constantes usadas durante el cálculo, etc. Es un documento relativamente técnico, para más información sobre este archivo de configuración véase al capítulo de Documentación, apartado *Main*, subapartado *Parametros.json.*

Seguidamente se obtendrán los datos necesarios para el cálculo. Se necesitarán cuatro datos para el cálculo: **demanda eléctrica horaria**, **precio de la electricidad por hora** para esas demandas, **irradiancia solar horaria** (solo relevante si se tienen paneles solares) y **temperaturas** (para predicciones principalmente).

La **demanda** eléctrica horaria se refiere al histórico de consumo que el usuario ha tenido. Este es uno de los dos datos más importantes para este problema. Se necesita conocer la demanda del usuario para calcular la batería que reducirá el coste, una batería que consuma en los valles y compense esta demanda en las puntas de precio horario. Este dato tan personalizado puede parecer imposible obtener. Un histórico de datos parece que solo se puede tener midiendo dichas demandas con antelación. Sin embargo, hay un organismo que también sabe las demandas personalizadas del usuario, la distribuidora, la cual ha estado midiendo y registrando estas demandas durante toda la duración del contrato. Solo se necesita pedir estos datos a la distribuidora. En el caso de Endesa, tiene una rama dedicada a la distribución energética, eDistribución, la cual tiene un servicio en su web de “descarga de curvas masiva” (eDistribución (Grupo Endesa)). Es decir, el dato exacto que se está buscando, un registro de datos horarios de demandas del usuario de la mejor exactitud y fiabilidad posible. Sin embargo, esta exactitud y personalización viene con la desventaja de que no podrá ser automatizado vía *software*, será un proceso manual que deberá hacer el usuario al menos una vez para configurar el sistema. Además, se deberán procesar estos datos descargados de eDistribución. Existe un módulo en el código dedicado a esto, con su respectivo apartado en el capítulo de documentación.

El siguiente dato será el **precio** horario del kWh. Es decir, cuánto costaba la electricidad a las horas de las demandas previamente obtenidas. Teniendo una demanda y un precio por dicha demanda se puede obtener fácilmente un precio total, simplemente multiplicando y sumando dichos valores. Y sabiendo cuándo la electricidad vale menos, se podrá cargar durante dichas horas la batería, para descargar en las horas más caras (tomando en consideración las horas durante las que la demanda del usuario es más alta). En España el organismo que fija estos precios es OMIE, el cual tiene en su web un histórico de precios públicos (OMIE (Operador del Mercado Ibérico de Energía)). El hecho de que estén en su web públicamente implica que estos datos podrán ser obtenidos por el programa, por lo que es un proceso transparente para el usuario. Sin embargo, este proceso

no es trivial, también tiene su propio módulo en el código, con su respectivo apartado en el capítulo de documentación.

El tercer dato, el de **irradiancia** solar hace referencia a la “cantidad de sol” que hay en la ubicación por hora. Es decir, la energía solar que llega a una superficie, medida en potencia por metro cuadrado. A partir de esta “cantidad de sol” y la cantidad de paneles solares que se quieren instalar se puede obtener la potencia que inyectan estos paneles en la red. Esto puede afectar al ciclo pues la energía generada por los paneles es esencialmente gratis, se puede cargar la batería con el excedente de energía solar y descargarla cuando sea necesaria. Para la obtención de este dato existen herramientas online para obtener un histórico de irradiancias solares, también cuenta con su propio módulo en el código y su capítulo en la documentación.

El último dato a obtener será la **temperatura**. Este dato no tendrá un efecto directo en el cálculo de optimización, pero los parámetros anteriores están fuertemente relacionados con este dato, especialmente los de demanda y precio de la electricidad. Por tanto, la temperatura será un parámetro fundamental para la predicción de datos futuros, siempre y cuando se pueda obtener una predicción de la propia temperatura previamente. Por suerte, existe una gran cantidad de herramientas online de predicción de temperatura, así que esto no será un gran impedimento. Pero aun así también necesitara su propio módulo en el código, con su respectivo apartado en la documentación.

Una vez se tienen los datos el siguiente paso es emparejarlos, relacionarlos para que cada dato corresponda con la misma hora. Si no se emparejan y alinean correctamente, unos datos no se corresponderán a la misma hora que el resto para hacer el cálculo, siendo erróneo desde el mismo planteamiento. A destacar la mezcla de datos internacionales y los “nacionales”, pues los datos de tanto eDistribución como los de OMIE estarán afectados por el cambio horario (cambio entre UTC+1 y UTC+2 en verano e invierno respectivamente) (Consejo de la Unión Europea (Council of the European Union)). Esto hace que haya un día del año que tenga 25 horas y otro 23. Se deberán reajustar los datos para acomodar esta información. Además, se deberán implementar medidas de validación de datos, tanto al obtenerlos como al combinarlos. Se dedica un módulo del código para toda esta parte de alineado y validación, así como un apartado dentro del capítulo de documentación en el que se profundiza más en la solución elegida. Al final esta parte genera un bloque de datos validados y correctamente emparejados, listo para ser usado por las siguientes partes del código como entrada principal. La Figura 2 es la síntesis visual de este subapartado.

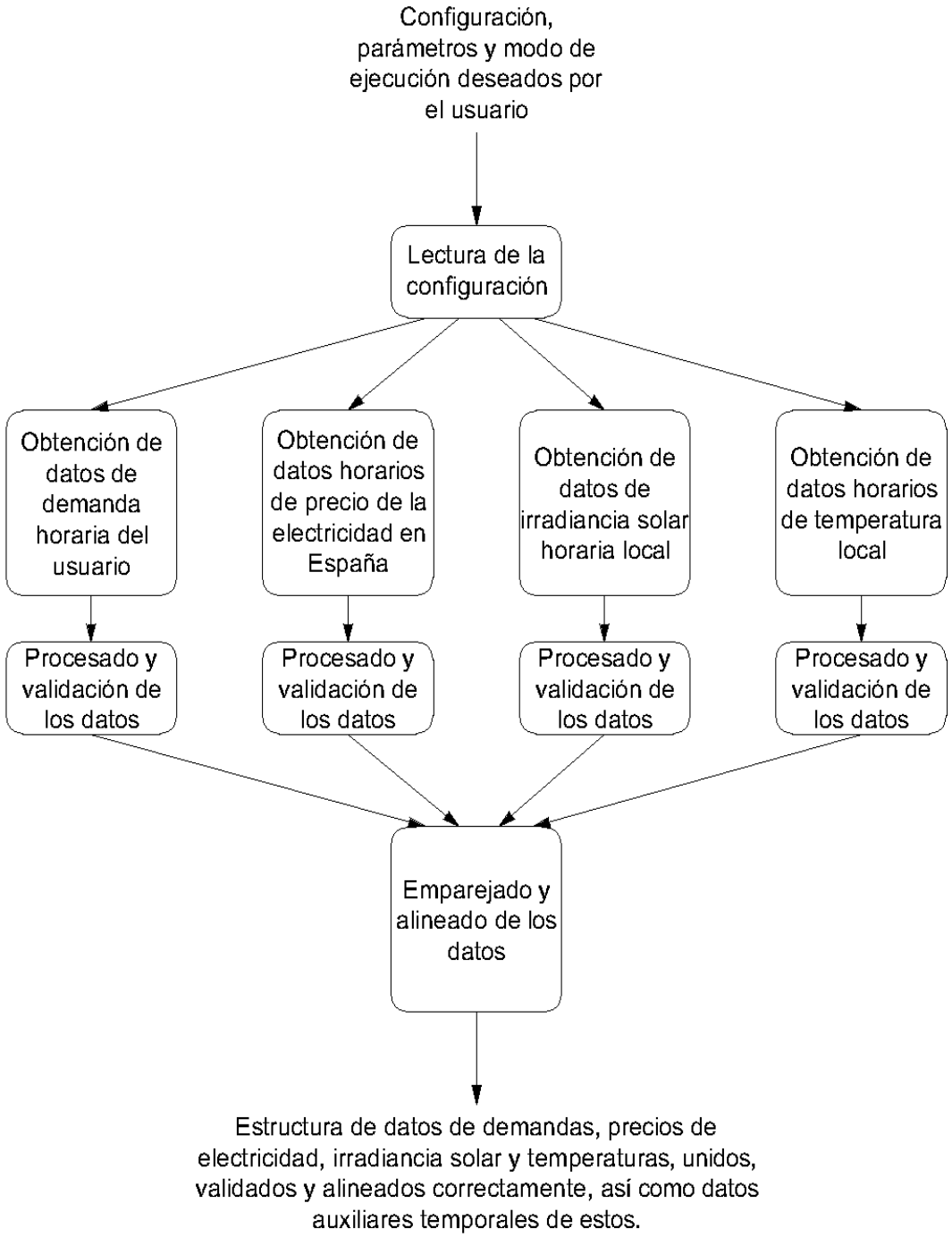


Figura 2. Flujo del código de la Parte Común.

Modo Histórico

Si se desea ejecutar un cálculo para elegir la batería óptima para los datos específicos del usuario entonces se deberá ejecutar el Modo Histórico a continuación de la Parte Común. Este modo toma las salidas de la Parte Común, es decir, un histórico de datos alineados y validados, y listos para ser procesados. Será objeto de este modo dicho procesado (cálculo).

Al igual que en la Parte Común, no se profundizará en el código usado, solo se seguirá la lógica empleada y los resultados obtenidos. Si se desea conocer en detalle el código se puede encontrar esta información en el capítulo de la documentación. Se referenciarán en esta parte principalmente los apartados de cálculo y presentación de datos. También se adjunta junto a la memoria el código fuente debidamente comentado y documentado.

Esta parte comienza tomando los datos que proporcionó la Parte Común. El objetivo será darle al usuario una **guía de qué batería debería comprar** para su situación. Sin embargo, esto no es tan sencillo como simplemente dar un número, pues esto es un problema de optimización esencialmente económico. La solución dependerá del mercado. Es decir, mientras más caras sean las baterías en el mercado, menos rentable será este sistema de comprar una batería para desplazar el consumo a las horas más baratas. No se podrá justificar el sistema si la batería es simplemente demasiado cara. Lo mismo ocurre al revés, si se está en una situación donde el mercado es favorable se podrá justificar la compra de una batería más grande para hacer el sistema más efectivo pudiendo almacenar más “energía barata”.

Por tanto, el *software* deberá calcular una capacidad de batería óptima para un rango de precios de mercado. Es decir, resolver el mismo problema usando los mismos datos de entrada, pero para distintos precios de mercado. Con esto se podrá generar una gráfica, una que relacione los precios de batería en el mercado con la capacidad de batería óptima para cada uno de ellos. El usuario deberá hacer un estudio del mercado de baterías, y obtener un precio del kWh en su situación concreta. Con ese dato podrá entrar en la gráfica desde el eje de abscisas, obteniendo en el de ordenadas la capacidad de batería recomendada para ese precio de mercado.

Se ha de considerar también la **relación entre la parte matemática y la parte física** del problema. El código resolverá un problema puramente matemático, existiendo así casos extremos, tanto de baterías enormes, potencialmente de cientos o incluso miles de kWh, así como valores de minúsculos, del rango de mWh. Estos valores obviamente no tendrán sentido físico, deberán ser “filtrados”. Pero tampoco es buena práctica ocultar información en una

herramienta que debe precisamente aportar esa información al usuario. Por tanto, se llega al compromiso de no dar solo una gráfica, sino dos. La primera tendrá el problema matemático íntegro, con los valores extremos representados gráficamente. Por la gran diferencia de valores esta gráfica no aportará mucha información, los valores útiles estarán "compactados" en muy poco espacio, pero aportara el contexto y el panorama general del cálculo. Para solucionar el problema del "compactado" de los datos útiles se incluirá entonces la segunda gráfica, la cual filtrará los valores previos, eliminando los valores extremos y dejando solo los datos centrales, los más razonables y con sentido físico. Será esta segunda gráfica la que el usuario consultará, la que verdaderamente aporte los valores y la información requerida.

Pueden ocurrir tres cosas una vez se tiene tanto la gráfica como el precio de mercado.

- **Batería de tamaño medio**: Será el caso más común, los datos que contempla la segunda gráfica, la de los datos filtrados. Se tienen datos tanto eléctricos como económicos nominales, y el *software* devuelve por tanto una solución normal, una batería de tamaño medio. Como referencia, se considera una "batería normal" para uso doméstico una de no más de unos pocos kWh, y no menos de 1 kWh (aunque podría ser factible en algunos casos ir hasta incluso 0,5 kWh para soluciones más económicas, este es el rango de batería de ácido de un coche, aunque en la inmensa mayoría de casos no será recomendable). Y en cuanto a precio depende del mercado, pero para ese rango de capacidades no se debería superar el rango de los 1000 o 2000 euros (aunque si se tiene un consumo particularmente alto el *software* puede considerar adecuada esta inversión), no mereciendo la pena soluciones de menos de 100 euros.

- **Batería muy pequeña**: Uno de los casos extremos que la primera gráfica refleja, pero la segunda ha descartado por no tener sentido, una batería en el rango de pocos Wh, o incluso mWh. Se dará este caso cuando el precio de mercado de las baterías sea muy alto y/o el consumo sea muy bajo. Se recomienda revisar el estudio de mercado, es posible que se esté buscando una batería de muy alta calidad, excesiva para este proyecto. Pero, si se ha confirmado que la entrada de datos es correcta, entonces se puede concluir que no merece la pena instalar una batería en primer lugar si se está en este rango. Simplemente no será rentable.

- **Batería muy grande**: El caso contrario al anterior, el otro extremo, baterías de decenas o cientos de kWh. Esta vez el *software* recomienda una batería excesivamente grande. En principio no será un problema, simplemente elegir la batería más grande que se esté dispuesto a instalar (que sea menor que lo recomendado por el *software*). Sin embargo, este rango está filtrado en la segunda gráfica por un motivo. Es muy posible

que se deba revisar el estudio de mercado, muy probablemente se está siendo muy optimista en el precio de la batería elegido, o se está eligiendo una de muy mala calidad, con los riesgos asociados a esta elección. En condiciones normales es muy poco común que se tenga un consumo tan alto y un mercado tan favorable como para justificar la instalación de una batería de tal capacidad.

Una vez se ha definidas las características y requerimientos que tendrá la solución buscada, se puede plantear su cálculo. Ya se ha definido que se deberá hacer un cálculo para un amplio rango de valores de mercado, pero aquí ya se encuentra el primer problema. Se necesita no solo definir el rango, sino además discretizarlo (definir puntos en dicho rango, no se puede hacer un cálculo "continuo"). Para la definición del rango se le pedirá al usuario que defina un valor tipo en la configuración, y a partir de ahí se puede definir un **rango representativo**. En cuanto a la **discretización** del rango no es tan trivial, mientras más puntos se elijan más precisión tendrá el cálculo. Pero la inclusión de más puntos implica más tiempo de cálculo. Se debe hacer un balance entre precisión y tiempo. Sin embargo, se han definido dos gráficas, una "general" (la que incluye todo el rango, incluso los datos extremos), y otra "detalle" (la que filtra este rango a los que tienen sentido físico). Puede parecer que esto será problemático pues hay que discretizar dos rangos, pero en realidad es una ventaja, se puede fijar una precisión de cálculo distinta para cada gráfica. Es decir, ya que por definición la gráfica "general" solo tiene el objetivo de mostrar la tendencia general se puede sacrificar precisión en pro de velocidad. Se pueden espaciar más los puntos de cálculo, dando a una densidad de puntos y cálculo relativamente baja. Caso contrario ocurre en la gráfica de "detalle", se puede emplear el tiempo ahorrado en la gráfica previa para obtener aquí un cálculo más preciso, con mayor densidad de puntos. Es en última instancia el usuario quien decide qué precisión y velocidad se desea. Se pueden ajustar estos parámetros desde el fichero de configuración. Se recomienda usar precisión baja y alta velocidad para cálculos preliminares, y una mayor precisión, aunque más lenta una vez se vaya a calcular el problema definitivo. Solo se deberá hacer este cálculo definitivo una vez, se considera un buen compromiso el sacrificio en velocidad a favor de un resultado final preciso.

Una vez se tienen los puntos a calcular definidos, y la Parte Común ha aportado los datos, se puede proceder a la resolución del problema. Se deberá resolver un problema de **optimización por cada punto** definido anteriormente. El cálculo consistirá en tomar todos los históricos de datos de demandas horarias del usuario, precio horario de la electricidad, y potencias solares horarias (en caso de que se quiera instalar y/o ampliar paneles solares). A estos datos se les suma la capacidad de la batería que calculará el problema, estando definido el coste del kWh por el punto que se esté calculando en cada momento. Se debe optimizar el coste, es decir, minimizarlo. Por tanto, la fórmula a optimizar deberá dar un coste, un número.

Antes de plantear la ecuación se debe entender como “funciona” el coste en este problema. El coste viene de dos fuentes, una es la inversión inicial que hay que hacer para comprar la batería (en función de capacidad de la batería y del coste del kWh), y la otra es pagar la electricidad consumida (la demanda total, suma de consumida, de la batería y paneles, multiplicada por el precio de la electricidad, el cual dará un precio por cada hora, que se deberá sumar para dar un total). También se debe tener en cuenta que en funcionamiento normal el sistema estará en funcionamiento varios años (según lo que indique la garantía del fabricante), periodo que tendrá el sistema para amortizarse. Para el cálculo no sería correcto esperar que se amortice enteramente solo durante los días en los que se tienen datos, pero en su lugar fraccionar el coste a amortiguar para que se iguale al caso real. La forma más fácil e intuitiva de entender y aplicar este concepto es definir un plazo de amortización en años, y dividir el coste de la batería en el número de horas de este periodo. De este modo ya se estará trabajando en la misma escala que el resto de parámetros del cálculo, haciendo las veces de costo fijo mientras que el consumo eléctrico representaría el costo variable. Se tiene entonces así el contexto necesario para plantear la ecuación a optimizar (minimizar):

$$\text{Coste} = \sum \left[(P_{\text{casa}} + P_{\text{batería}} - P_{\text{solar}} \cdot \text{Coef}_{\text{solar}}) \cdot \text{Precio} \; + \; \frac{C_{\text{bat usable}} \cdot \text{Precio bat} \cdot \text{Hora calc}}{\text{Nº horas total}} \right]$$

Ecuación 1. Definición de ecuación a optimizar (minimizar)

Para aclarar que significa cada término de la Ecuación 1:

- **Demanda de la casa (P_{casa}):** Potencia horaria demandada por el usuario.
- **Precio (*Precio bat*):** Precio horario de la electricidad.
- **Paneles (P_{solar}):** Potencia horaria generada por los paneles solares.
- **Capacidad de la batería ($C_{bat\,\text{usable}}$):** Capacidad en kWh de la batería.
- **Demanda de la batería ($P_{batería}$):** Potencia horaria demandada por la batería (ciclo de la batería).
- **Coeficientes solares ($Coef_{solar}$):** Variable auxiliar, porcentaje de utilización de los paneles solares.
- **Precio bat:** Precio del kWh de la batería.
- **Número de horas total (*Nº horas total*):** Número de horas total mínimas que el fabricante garantiza que la batería será funcional.
- **Horas de cálculo (*Hora calc*):** Horas de que se evaluarán y calcularán en el problema (este parámetro es un vector, para igualar al primer término).

De un vistazo se pueden diferenciar dos elementos que se han introducido ya. Uno es el sumatorio, pues estos términos están todos en escala horaria, se necesita un coste total, es decir, sumar todos los costes de todas las horas. El segundo elemento es el signo más, dividiendo la fórmula en dos claros términos,

uno para el “coste fijo”, otro para el “coste variable”. El primer sumando representaría al coste variable, es decir, al precio pagado por la energía consumida neta (variando el precio por kWh consumido cada hora). Más consumo, más coste. El segundo término es el análogo a un costo fijo. Reparte la amortización durante el periodo de cálculo, de una forma que sea compatible con el sumatorio. Se pueden analizar así los sumandos por separado para entender mejor la ecuación. El objetivo final es **minimizar el coste total**.

Si se quiere más información de esta ecuación en la lectura del subapartado de *calculo_CPU*, en el apartado de cálculo de la capacidad de la batería del capítulo de la documentación. Se entra en más detalle en este cálculo, a un nivel de abstracción más bajo. Pero, retomando el flujo, el primer término hace un sumatorio de potencias y luego multiplica esta potencia neta consumida por el precio. Más demanda o más precio se traducen en más coste, **precio variable**. Conceptualmente sencillo, pero hay cuestiones a destacar.

La primera y más llamativa esa variable de coeficiente. Simplemente representa la capacidad de desconectar a voluntad los paneles solares, será un valor entre 0 y 1 que representa qué utilización tienen los paneles. Se podría pensar que este valor no es necesario pues se podría volcar el excedente de energía solar a la red, e incluso venderla. Y si bien esa es la solución más común actualmente, cabe la posibilidad de que no siempre se pueda usar en España. Esto es debido a la enorme cantidad de potencia instalada de energía solar, algo positivo pues es energía renovable, abundante y barata, pero también es por naturaleza inestable (ENTSO-E (European Network of Transmission System Operators for Electricity), 2017), lo que da lugar a inestabilidad en la red, pudiendo causar inestabilidad a nivel nacional y el posterior colapso de la red si no se soluciona a tiempo (Red Eléctrica de España (REE), 2025). Esto viene secundado por la existencia de circuitos de grandes fabricantes y vendedores dedicados a impedir el vertido a red (Huawei Technologies Co., Ltd., 2022). Se incluye esta posibilidad como un coeficiente. Si se puede volcar y vender entonces se podrá mantener este coeficiente siempre a 1. Pero si no se puede garantizar, se tiene integrada en la ecuación la posibilidad de usar valores menores de la unidad.

La segunda cuestión que surge es relacionada a demanda de la batería. ¿Cuál es esta demanda? Aparece así una de las variables a optimizar, el ciclo de la batería. Recordatorio: se debe obtener el ciclo de la batería que minimice el coste. Conceptualmente es simplemente “cargar en las horas baratas y descargar en las horas caras”. Matemáticamente no es tan sencillo y es el núcleo de este problema. Es necesario el uso de complejos algoritmos para determinar este ciclo óptimo. Referirse al subapartado de *calculo_CPU* mencionado previamente para más información de este algoritmo.

En cuanto al segundo término, hace las veces de **coste fijo**. Aparecen otras dos particularidades. La primera es que para poder adaptarlo al formato de sumatorio las horas serán un vector, dando así entonces un precio fijo para cada hora (todos iguales, repartidos equitativamente). Se sumará este coste fijo horario a la variable del término anterior.

La segunda es la variable de la capacidad de la batería. Análogo al término previo, esta es una variable que el algoritmo deberá optimizar para minimizar el coste. Esta capacidad limitará al ciclo visto previamente. Es decir, a mayor batería más efectivo podrá ser el ciclo y mayor potencial de ahorro, pero también más cara será la batería y mayor inversión. Se debe puntualizar que para el cálculo se usa una capacidad de la batería efectiva, solo se usa un porcentaje de la capacidad total con el objetivo de maximizar la vida útil de la misma (Joselyn Stephane Menye, 2025). La profundidad de este ciclo dependerá de la tecnología de la batería. Para baterías de litio, la tecnología más común para baterías actualmente, no suele ser recomendable usar una profundidad mucho mayor del 60% (entre el 20% y el 80% de la capacidad máxima). El fabricante suele facilitar este dato, es quien mejor sabe qué profundidad es la óptima para sus baterías, pero hasta tener elegido una tecnología y fabricante se debe estimar este valor.

Es especialmente destacable esta variable de la capacidad de la batería en este Modo Histórico, pues será el dato buscado para imprimir en la gráfica resultado final. A cada punto de precio del kWh le corresponde su capacidad de batería óptima. Uniendo los puntos que se van a calcular con estos valores calculados se puede obtener la gráfica deseada de guía de compra de la batería.

Recapitulando, la Figura 3 incluye una representación gráfica de este flujo del Modo Histórico.

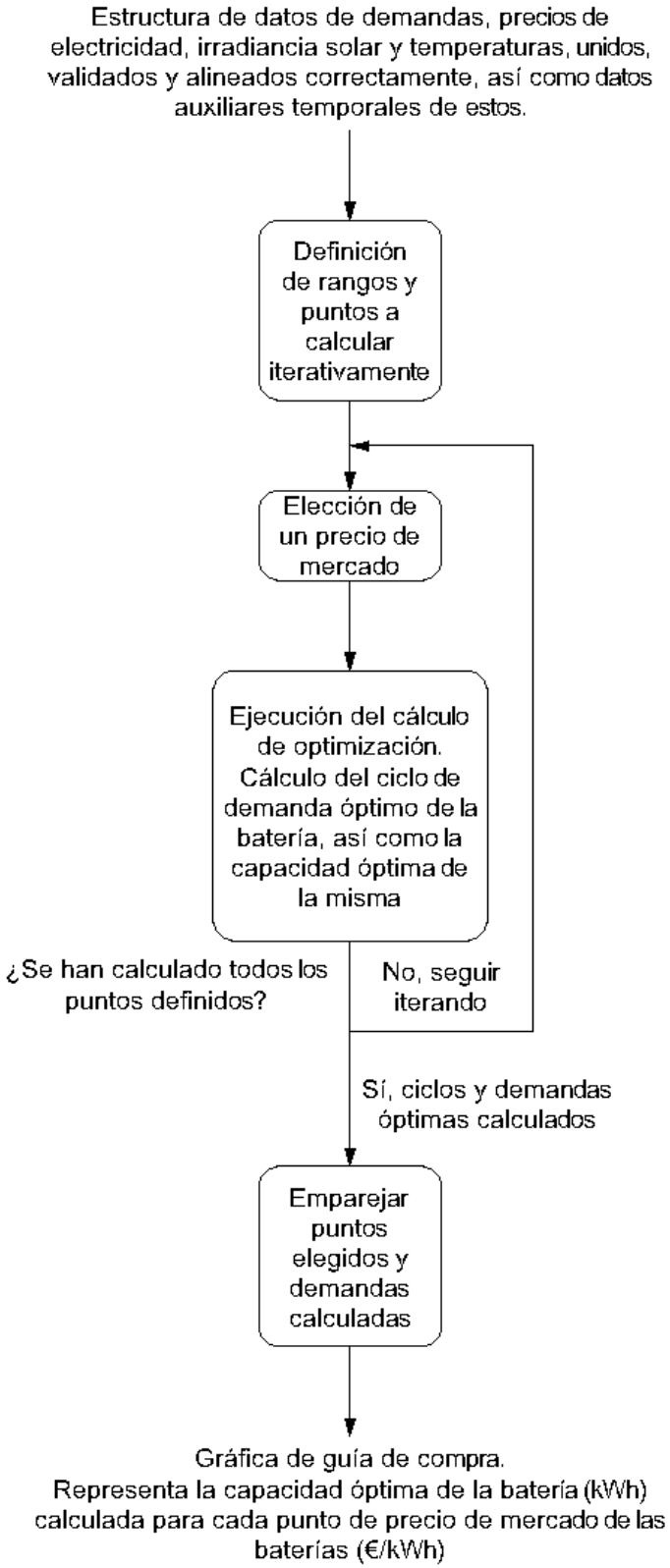


Figura 3. Flujo del Modo Histórico.

Modo Futuro

Si se eligió el modo de cálculo futuro es porque ya se tiene instalada una batería y se desea predecir el ciclo para mañana de la misma. Este resultado no tiene como objetivo ser leído e interpretado por un humano, pero en su lugar su principal objetivo es **enviarlo al controlador de la batería**, un ordenador, para que pueda usar estas directivas para funcionar durante todo el día entero. Por tanto, no se busca una gráfica "bonita", pero en su lugar un fichero de texto con datos "en bruto". Se usa un fichero de texto plano (*.txt*) porque es el formato más universal, capaz de ser leído por la inmensa mayoría de controladores, incluso los menos potentes.

Sin embargo, si bien el resultado primario será este fichero de texto con datos para el controlador, se debe incluir una salida "humana" de estos datos, tanto para el desarrollo del código para comprobar que se progrese en la dirección correcta y sin errores. Una vez acabado el desarrollo del código, el usuario puede requerir una comprobación de que el sistema funcione correctamente. Esto es tan simple como **representar gráficamente los datos** que se mandan al controlador, siendo este el resultado "secundario", los mismos datos, pero en forma de gráfica, únicamente para información del usuario.

La generación de este ciclo para el próximo día tiene implícito una predicción de datos para dicho próximo día. Este problema se puede abarcar de varias formas, cada una de diferentes grados de complejidad. En general, este es un problema difícil, que incluso a grandes organismos les cuesta resolver, pues implica modelos estadísticos y/o matemáticos muy complejos y avanzados, con su respectivo coste computacional asociado. No se disponen de las herramientas para un cálculo de tal nivel, así que en caso de emplear una de estas soluciones estadísticas se deberá usar una versión mucho más simplificada, y como consecuencia, menos fiable.

Sin embargo, esto ha sido el panorama tradicionalmente, pero en los últimos años el paradigma ha cambiado, existe una herramienta más. La **inteligencia artificial** (Brown, y otros, 2020). No es objeto de este texto explicar cómo funciona esta herramienta, pues iguala, si no supera, la complejidad de las herramientas clásicas, pero como simplificación, la inteligencia artificial está especializada en el reconocimiento de patrones. Se le pueden dar datos históricos particulares del usuario y entrenarla con estos datos, de modo que se obtendrá un modelo de IA entrenado y especializado únicamente para los datos particulares del usuario. Con la suficiente cantidad de datos podrá ser capaz de capturar los patrones inherentes en los datos eléctricos, tanto de demanda del usuario principalmente, así como del precio de la electricidad y ciclo solar en segunda instancia.

Pero al igual que la IA tiene grandes virtudes, y grandes ventajas para este problema en específico, también tiene sus respectivas desventajas, casi igual de grandes que sus contrarias. La primera y más importante es la cuestionable fiabilidad. Esto es un problema intrínseco de las inteligencias artificiales, y en parte también debido a la falta de madurez de esta herramienta, pues lleva pocos años siendo usada comercialmente. Simplificando este inconveniente, por cómo funciona la IA, esta genera en cada activación una respuesta única y diferente cada vez, lo que significa que algunas de estas generaciones simplemente serán incorrectas. A este efecto se le conoce como “alucinaciones” (Ji, y otros, 2023). Y no existe forma sencilla de separar estas generaciones incorrectas de las correctas, naciendo así el mayor problema de esta herramienta, su baja confiabilidad. El segundo inconveniente es su coste computacional. Si bien no es un problema tan grave como el primero, ya que tiene una solución clara (mejor y más costoso hardware), es poco eficiente usando el hardware disponible. El entrenamiento de incluso un modelo de IA sencillo necesita costosas tarjetas gráficas de varios cientos o incluso miles de euros, por lo que no es compatible con ordenadores más antiguos o poco potentes (aunque se ha de romper una lanza a favor, una vez entrenado dicho modelo no es tan caro de ejecutar). Se le suma además el elevado consumo eléctrico que conllevan estas tarjetas gráficas (Strubell, y otros, 2019). Hay una cierta ironía en que el problema global a resolver tenga el objetivo de gastar menos, mientras que por otro lado la IA es poco eficiente (pero nuevamente, en realidad solo el entrenamiento es caro, la ejecución del modelo no es muy costosa eléctrica y computacionalmente). Y, por último, pero no menos importante, la tercera desventaja son los datos. Las IA necesitan vastas cantidades de datos para ser entrenadas, así como datos de control de muy buena calidad para verificar que el entrenamiento se efectúe correctamente. La Parte Común que precede a este Modo Futuro es capaz de generar cientos de miles de datos, que puede parecer que cumplen con este calificativo de “vastos”, pero no, esta cantidad no es ni remotamente cercana a la cantidad de información que necesita una IA para ser entrenada (Kaplan, y otros, 2020). Una tarjeta gráfica moderna es capaz de procesar esa información en segundos. Ya que no se disponen de tantos datos de esta calidad, se deberá bajar la calidad de los datos para subir la cantidad, lo que agrava aún más la primera desventaja. Se abarca en más profundidad la solución aplicada para este impedimento en apartado de la IA de la documentación.

Vistas tanto las ventajas como las desventajas de la IA, se decide que la capacidad de personalizar un modelo a los datos del usuario, con la capacidad de predecir los patrones específicos del usuario es lo suficientemente valioso como para que se sobreponga a las desventajas, así que se entrenarán modelos de IA previamente usando estos datos. El proceso de entrenamiento es algo bastante complejo y que no tiene cabida en este capítulo, se aborda este aspecto en el apartado dedicado a la IA de la documentación, como ya se ha mencionado. En dicho apartado también se presenta la solución al problema de no ser compatible con hardware más antiguo, que es simplemente que si no se cumplen los requisitos, se incluye en el código un modelo matemático clásico de predicción de series, mucho más eficiente, pero más genérico.

Con esto, queda definido el destino y se vislumbra el viaje, queda por iniciarlo. Lo primero que se hace tras recibir los datos históricos que genera la Parte Común es guardarlos y completarlos más. Se obtienen los datos de la fuente original, a mayor cantidad, mejor. Así se minimiza el problema de la poca fiabilidad de la IA (mientras menos datos se tengan que generar, menos datos estarán mal). El caso de la temperatura es especial. La temperatura será una de las entradas que use el modelo de la IA, pues tanto la demanda del usuario, como el precio de la electricidad, así como la potencia solar están fuertemente ligadas a la temperatura (International Energy Agency, 2025) (Mosquera-López, y otros, 2024) (Walker, y otros, 2022). Pero si la temperatura es una entrada de la IA esto significa que dicha IA no podrá generarla. Se recurre entonces a webs meteorológicas que pueden dar una predicción de este dato, solventando el problema y en el proceso obteniendo mejores datos que los que podría haber generado la IA.

Una vez obtenida la mayor cantidad de datos posibles de las fuentes originales, y aplicadas las mismas alineaciones y verificaciones que se aplicaron en la Parte Común, llega el momento de usar la IA. El entrenamiento de los modelos es un proceso aparte y manual (referirse al capítulo ya mencionado de la documentación), en este modo solo se usarán estos modelos ya creados. La IA no solo completará hasta el presente los datos que no se pudieran obtener de la fuente, sino que además seguirá prediciendo hasta unos pocos días en el futuro. Generará así los tres datos eléctricos necesarios para el cálculo de optimización para días en el futuro: demanda del usuario, precio de la electricidad, y potencia solar.

Una vez se tienen los datos se puede usar el mismo algoritmo de optimización que se empleó en el Modo Histórico, solo que esta vez se resolverá un problema mucho más sencillo. Solo se optimizarán unos pocos días, y no será necesario hacer el cálculo iterativo propio de dicho Modo. Además, la capacidad de la batería ya estará fijada, así que solo le queda una única variable por optimizar, el ciclo de dicha batería. Para conocer en qué consiste este cálculo de optimización referirse al Modo Histórico, o bien al capítulo de cálculo de la documentación si se requiere un comentario más técnico.

Como apunte, se ha observado que OMIE actualiza su web con datos del próximo día por la tarde (hora peninsular), así que, si se ejecuta este cálculo en las últimas horas del día, la primera parte de este modo, la de obtener de la fuente la mayor cantidad de datos posibles, será capaz de obtener los precios reales que se usarán al día siguiente. Se mejora así la fiabilidad y calidad de los datos, minimizando el mayor inconveniente de la IA aún más.

Una vez ejecutado el cálculo de optimización ya se obtiene el ciclo que deberá seguir la batería el día siguiente, se le puede pasar la planificación a su

controlador en forma del archivo de texto simple. Opcionalmente, si el usuario lo requiere se puede crear una gráfica con estos datos, para que pueda observar el ciclo calculado y verificar su correcto funcionamiento.

A modo de resumen, se puede observar el flujo de este modo gráficamente con la Figura 4.

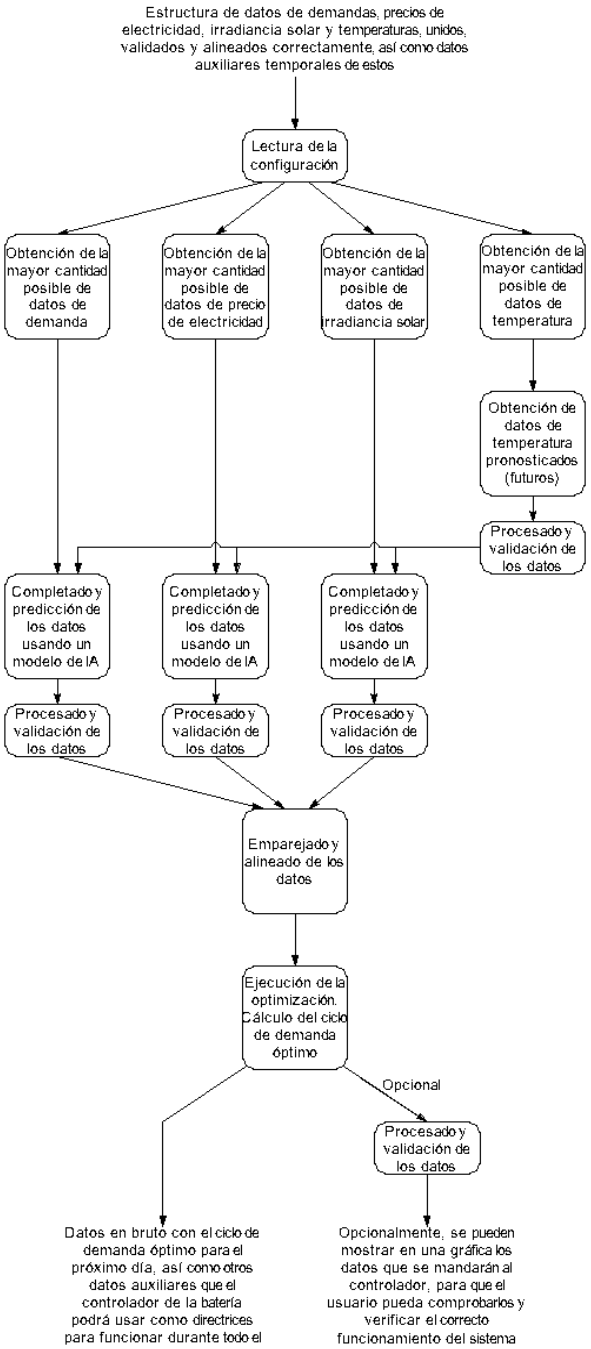


Figura 4. Flujo del Modo Futuro.

Visualización y resumen del flujo del código

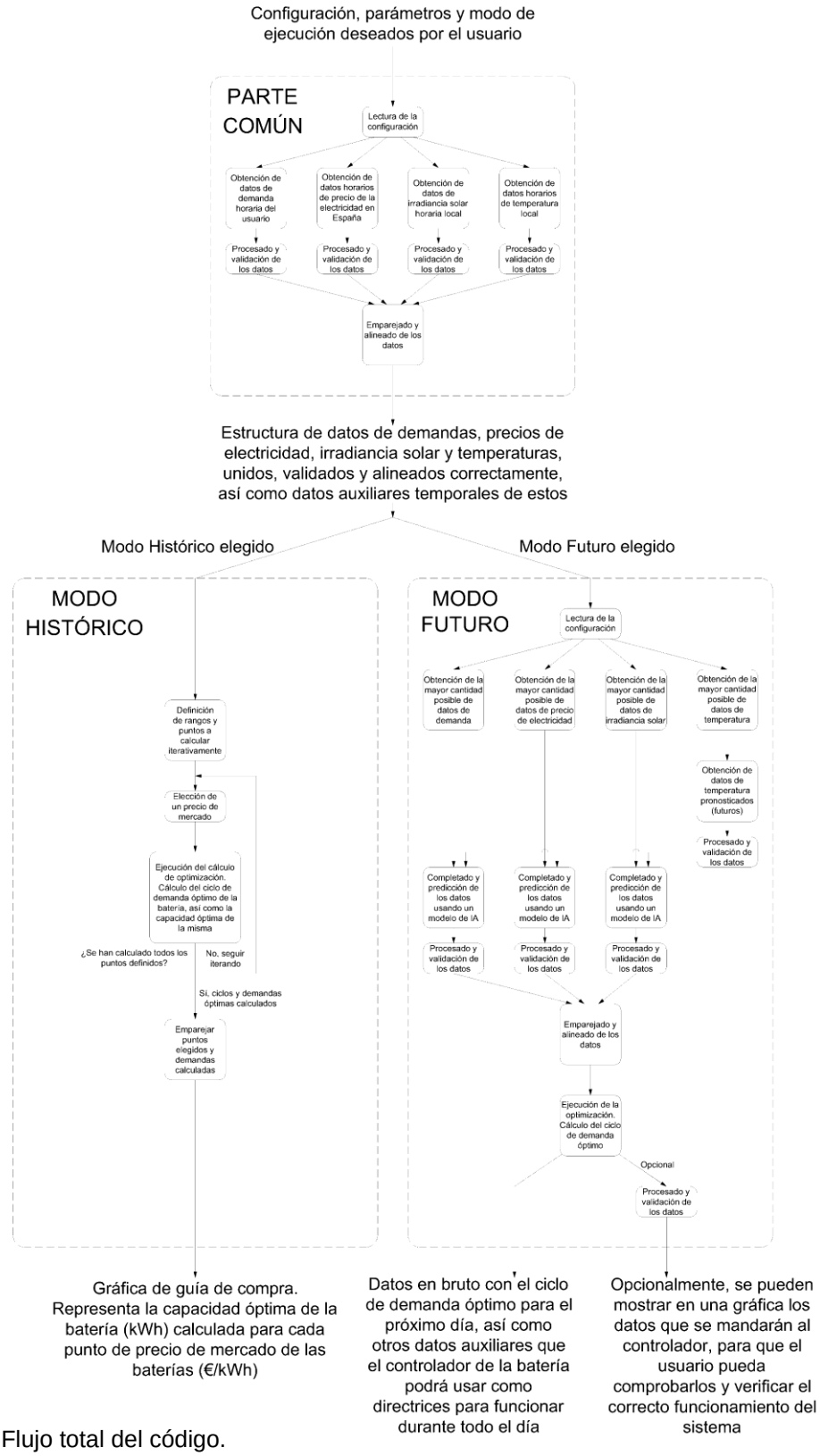


Figura 5. Flujo total del código.

Completando los esquemas que se expusieron en el anterior apartado, se pueden combinar en un único esquema que puede utilizarse de referencia durante todo el código, tal y como se ve en la Figura 5. A continuación se presenta un texto con una breve explicación de dicho flujo a modo de resumen rápido.

Flujo del código. Cuando el usuario configure y defina los parámetros necesarios en el archivo de parámetros, decidirá qué modo quiere ejecutar, el histórico para obtener una recomendación de compra de la batería o el diario para calcular el ciclo de dicha batería ya comprada e instalada. En cualquier caso, el inicio es el mismo para ambos, se obtienen los datos históricos de demanda, precio, solar y temperatura, y se alinean y verifican. Con estos datos llega el momento en el que el flujo se ramifica según la decisión inicial. Si se selecciona el modo histórico se usarán estos datos para calcular el problema de optimización y obtener la gráfica de recomendación de compra en función del precio de mercado (recomienda una capacidad según su precio por kWh que se pueda obtener en el mercado). Si en caso contrario se selecciona el modo diario se continuará con la obtención de datos. Se obtiene la mayor cantidad posible desde la fuente original, y se completan con IA los que no se puedan obtener así, aplicando durante el proceso las mismas verificaciones y alineaciones que para los datos históricos. Se finaliza calculando el ciclo óptimo de demanda para el próximo día.

RESULTADOS

Una vez se comprenden las bases del programa, el siguiente paso es ejecutarlo y ver los resultados obtenidos. Por definición está diseñado para ser altamente adaptable a las condiciones particulares del usuario. Sin embargo, un buen análisis y comparación de datos requiere una base estable y común para poder ser comparados. Esto requiere que se prescinda de esta flexibilidad para este capítulo. Todos los cálculos y resultados presentados estarán basados en los mismos datos, y **la configuración siempre será la misma**.

Los datos usados para la demanda eléctrica serán datos personales, y la configuración estará basada en la misma instalación personal. Los datos más destacables serán:

- Potencia contratada: 4kW
- Ubicación, ciudad: Sevilla
- Periodo de amortización: 10 años

- Potencia máxima de carga/descarga de la batería: 1kW
- Capacidad usable de la batería (profundidad del ciclo): 65%
- Fechas de datos históricos: 01-03-23 a 02-03-25
- Paneles históricos a instalar de 10 m^2 (unos 2.5kWp)

Una vez definida la configuración, se ejecuta el programa. Ya que existen dos modos se dividen los resultados en dos apartados, uno para el modo histórico, otro para el modo diario.

Resultados del Modo Histórico

A modo de recordatorio e introducción, el **Modo Histórico** calcula para un rango de precios la capacidad de batería más óptima para cada precio, así como la capacidad y los ciclos que justifican este precio. Estos ciclos normalmente quedarían ocultos para el usuario, pues no afectan a la decisión final de compra, y además la visualización de todos los ciclos que calcula el programa es virtualmente imposible para un humano. Sin embargo, puede ser de interés académico la visualización de algunos de estos ciclos, los más relevantes para el resultado final.

Además, si bien se ha dicho que toda buena comparación de resultados parte de una base común, eso implica que solo se podrá obtener un único resultado. Esto no es de interés para esta memoria, pues también se busca ver como se adapta a distintos datos. Se incluyen por tanto varios sub apartados, uno con los datos estándar, y otros con variaciones a estos datos, con fines de exploración académica.

Datos estándar

Se ejecuta el script usando el argumento necesario para pasar al Modo Histórico. Al ejecutarlo se muestra por consola información de depuracion, así como las primeras indicaciones de cálculos y valores calculados. Sin embargo, el cálculo es tan rápido que esta información tiene poca utilidad mientras se está ejecutando. Al final imprime esta gráfica, Figura 6:

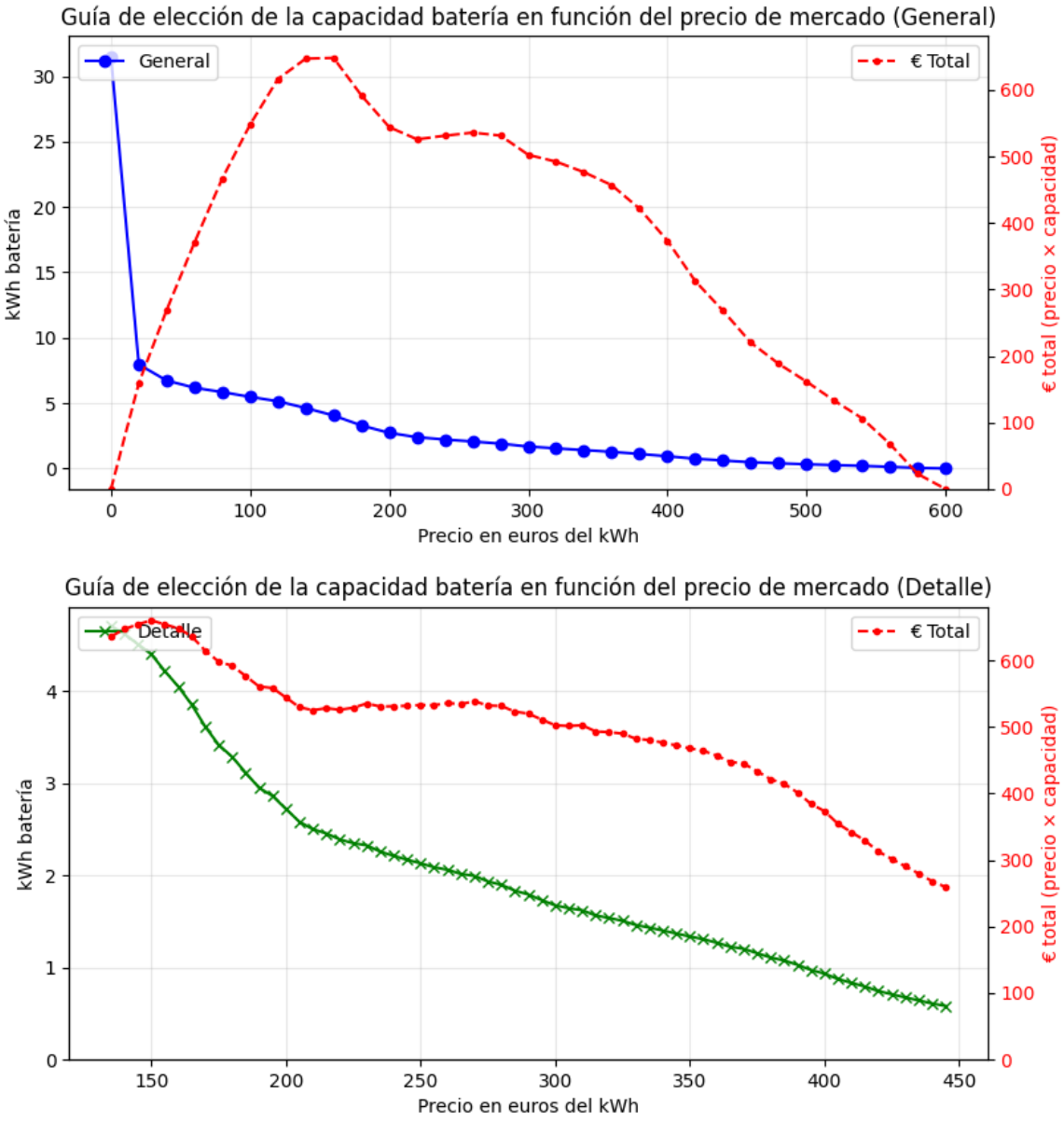


Figura 6. Resultados estándar del modo histórico.

La Figura 6 contiene toda la información necesaria que el usuario necesita para tomar una decisión informada sobre la compra de la batería que se quiere instalar, inclusive si no fuera rentable la compra. Para empezar a desgranar la información representada, lo primero que se ve es que hay dos gráficas, una "**General**" y otra de "**Detalle**". Como ya se dijo tanto en el flujo del código del capítulo previo, como también en su apartado correspondiente en el capítulo de documentación, en realidad no son dos gráficas, es la misma, solo que la segunda, la de detalle, es solo una sección de la primera, la general. Es decir, la

de detalle solo se centra en los datos más relevantes de la general. Por tanto, ya que muestran la misma información, lo que se aclare de una se aplica a la otra.

Lo siguiente que se ve es que hay dos líneas, ambas compartiendo el mismo eje de abscisas, con los precios en euros del kWh de las baterías (precio de mercado). Pero los ejes de ordenadas son distintos. A la izquierda, y siguiendo la representación de gráficas tradicional, el principal, la **capacidad óptima** de la batería para ese precio. La batería que se ha calculado que optimiza y minimiza el coste de la electricidad. Esta es la línea que el usuario necesita para tomar la decisión. La línea azul en la gráfica general, y la verde en la de detalle. Y ya que la de detalle por definición contiene el rango de valores más útiles, esta será la línea más útil para el usuario. Simplemente se debe estudiar el mercado, ver el precio en euros del kWh y elegir a partir de ahí. Por ejemplo, para un precio de mercado de 200 euros el kWh el programa ha calculado que la capacidad de la batería óptima está en los 2,5 kWh aproximadamente (incluyendo ya aquí el porcentaje usable de la capacidad total). Si las condiciones de mercado no fueran favorables, y el precio del kWh fuera el doble (400 euros el kWh), entonces el script recomienda apenas una batería de 1 kWh. Si fueran aún más caras entonces es posible que no sea rentable instalar este sistema, el precio de la batería sería prohibitivo para la demanda y los precios de la electricidad que se tienen.

Falta por hablar de la segunda línea, la línea roja. Esta en realidad no aporta información nueva, es algo que ya va implícito en la línea principal, pero aun así es útil tener este dato visualmente. Es el precio de la **inversión total**. Puede parecer que es un dato nuevo, pero no, pues previamente se ha hablado de tanto €/kWh como de kWh. Una simple multiplicación ya da esta gráfica. Siguiendo los ejemplos previos, 2,5kWh a 200 €/kWh da una inversión de unos 500 €, lo cual confirma la línea roja. También confirma el segundo ejemplo, 1 kW a 400 €/kWh son 400 € de inversión. Pero como ya se ha mencionado, ya que esta decisión es esencialmente una monetaria, es útil tener de un vistazo la inversión que se haría para ayudar a la elección. Como inciso, se muestra la inversión total, pero no el ahorro. Esta omisión del dato de ahorro es a propósito, porque si bien en este punto ya se tiene un dato de ahorro interno matemáticamente ideal, estará tan alejado del real, con todas las particularidades del cálculo real y las predicciones de datos, que simplemente sería un dato engañoso. Es más recomendable omitirlo en este punto y verlo posteriormente con más detalle.

A destacar la forma que hace la inversión recomendada (líneas rojas). En un principio es baja por los precios irrealmente bajos del kWh, y luego llega a un máximo una vez dichos precios empiezan a ser realistas, pero lo suficientemente bajos para que pueda justificarse la instalación de una gran capacidad (a mayor capacidad mayor será también la efectividad del sistema y mayor ahorro cada día). Pero a partir de ese máximo, el precio recomendado empieza a caer, pues los precios empiezan a ser cada vez más prohibitivos, lo que impide instalar

baterías tan grandes y efectivas, lo que a su vez reduce la capacidad de ahorro del sistema. Se confirma así la hipótesis inicial, mientras más grande sea la batería más capacidad de ahorro se tendrá pues podrá ser más efectiva en el sistema eléctrico (pero también más cara).

Ahora bien, ¿cómo obtiene el programa estos valores óptimos? ¿Qué solución de ciclos de batería está usando? Esto es una justificación de muchos datos, difíciles de visualizar y entender para un humano. Pero se pueden ver los dos casos concretos que han sido elegidos arbitrariamente de ejemplos.

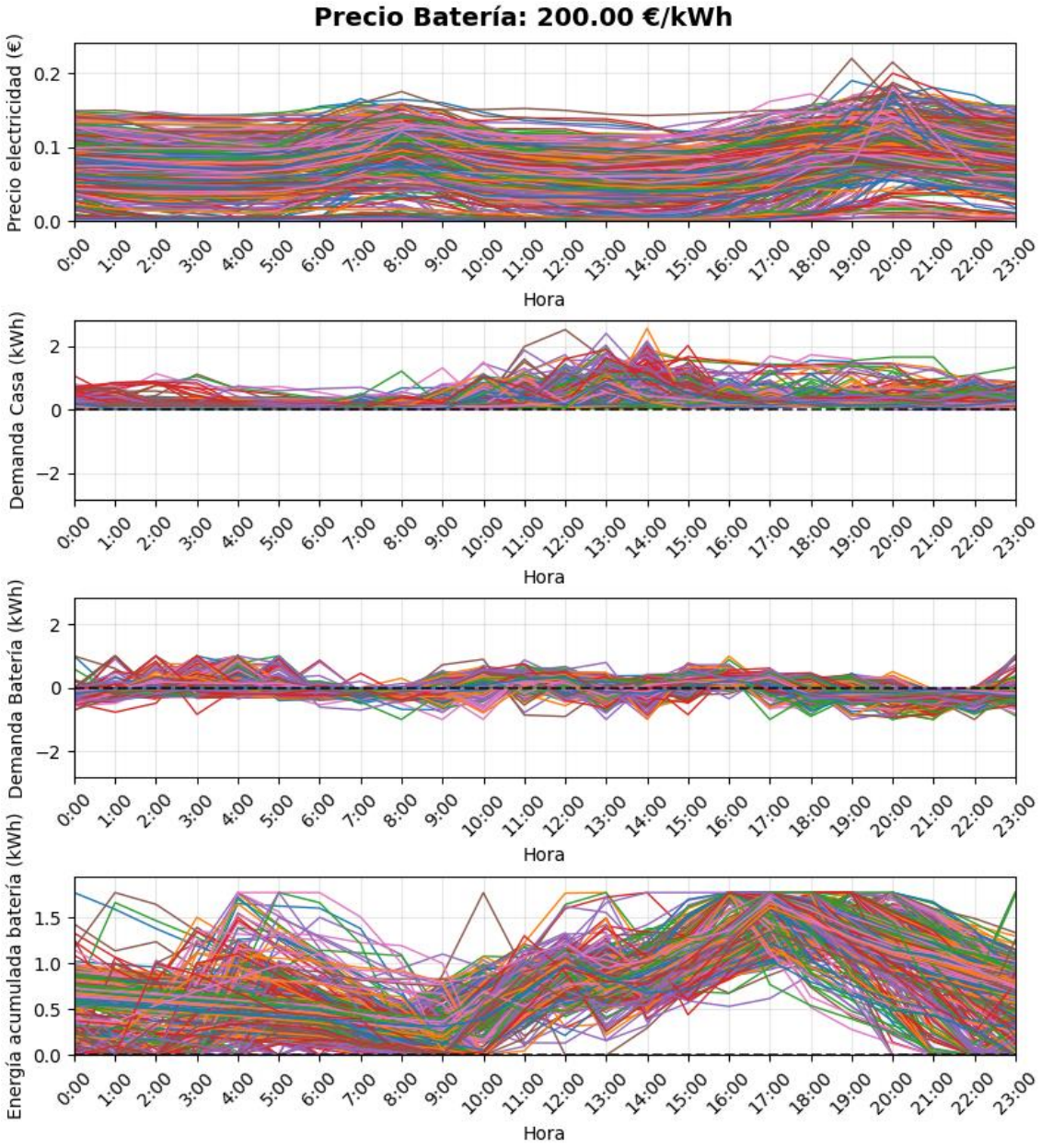


Figura 7. Ciclo calculado para precio de mercado de la batería de 200 €/kWh.

La Figura 7 es tal como se esperaba, muy caótica y confusa, da demasiada información. Resulta imposible diferenciar un día de otro. Pero se pueden ver las **tendencias generales**. Cada línea representa un día (datos de dos años, más de 700 líneas), variando los colores para aportar algo de orden dentro del caos. El eje de abscisas es común para todas las gráficas y representa horas. Unido al hecho de que cada línea es un día, estas gráficas representan los ciclos diarios de cada uno de los parámetros graficados. Siendo estos parámetros graficados los indicados en las etiquetas en sus respectivos ejes de ordenadas. La primera son los precios de electricidad de OMIE, si se va a su web se verán estos mismos datos de 2023 y 2024. Se representa en €/kWh. La segunda es la demanda de la casa en kWh. Este es el dato personal que se mencionó al inicio del capítulo, cada usuario tendrá el suyo propio. Se puede apreciar que las demandas son relativamente bajas, así que el *software* tendrá poco margen para generar ahorros. Se representa en kWh (cada hora, si se dividen es energía entre tiempo, potencia). La tercera gráfica es la buscada, el ciclo de potencia de la batería que ha calculado el programa. Esta es la gráfica que indica cuándo consume o cuándo descarga la batería, y a qué potencia. Se mide en kWh, al igual que la demanda de la casa. Nótese que la potencia de la batería está limitada por configuración a solo 1kW, así que se puede ver aquí claramente que ni en carga ni en descarga se supera ese valor.

Para entrar en más profundidad en esta tercera gráfica, lo primero que hay que destacar es ver qué significan los datos positivos y qué los datos negativos. Es simple, esto es potencia consumida por la batería. Un dato positivo es simplemente potencia consumida por la batería, se está cargando. Por el contrario, un dato negativo significa que la batería está cediendo potencia, descargando en la casa su energía acumulada. El ciclo varía mucho diariamente, pero se pueden ver dos tendencias generales. La primera es un ciclo de carga al inicio, en las horas de madrugada. Contrastando con las dos gráficas previas se ve que son las horas con los precios más bajos y con las demandas de la casa más bajas, por lo cual tiene sentido que cargue en este plazo. La segunda tendencia que mantiene aproximadamente todos los días es la descarga a las últimas horas del día. Caso contrario al anterior, son las horas más caras y con más demanda, así que esta solución vuelve a tener sentido intuitivamente.

Por último, la última gráfica muestra la energía acumulada de la batería. Este valor es simplemente el sumatorio acumulado de las potencias, así que no aporta información nueva. Sin embargo, esta forma de representar datos distinta a la anterior es útil para ver otros datos. Concretamente la capacidad máxima, la batería elegida. Esta gráfica muestra energía acumulada (kWh), así que el pico máximo de energía acumulada será la capacidad máxima (usable) de la misma. Como recordatorio, en la Figura 6, se podía ver que para 200 €/kWh se recomendaba comprar unos 2,5 kWh. Usando solo el 65% de dicha capacidad máxima quedan disponibles unos 1,63 kWh. Se puede ver que efectivamente, se llega a esta capacidad consistentemente, especialmente a media tarde, justo antes de empezar las horas pico de la noche (cuando mayor es la descarga), y

por tanto cuando mayor carga acumulada se tendrá. Ya se había podido deducir esta información con la gráfica previa, pero la forma distinta de presentar esta información puede ser más intuitiva en algunos casos.

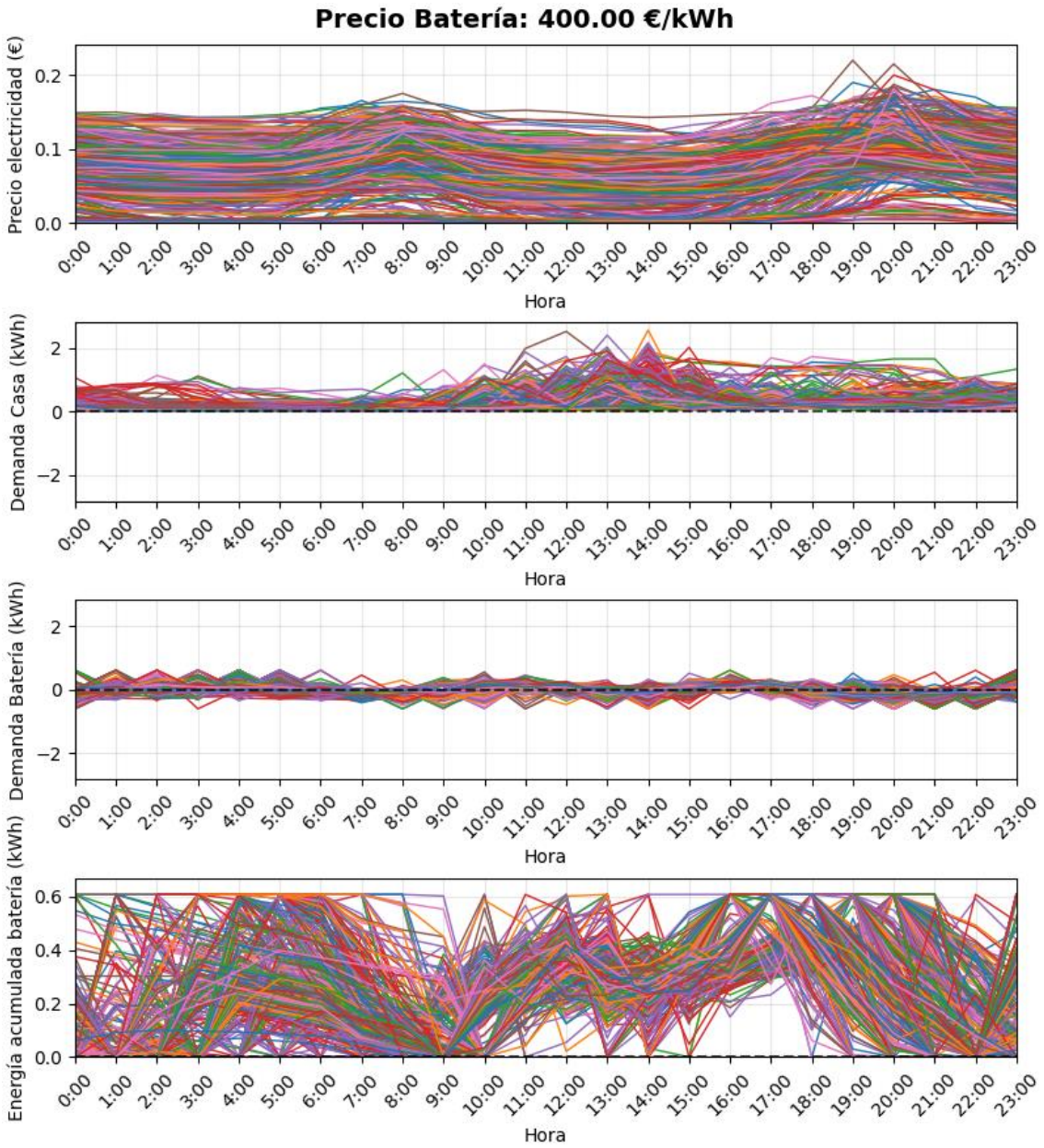


Figura 8. Ciclo calculado para precio de mercado de la batería de 400 €/kWh.

En cuanto al segundo ejemplo, el de los 400 €/kWh representado en la Figura 8, se puede ver una solución bastante distinta, especialmente en la energía acumulada de la cuarta gráfica.

Las dos primeras gráficas son datos así que permanecen constantes. La tercera, el ciclo de la batería ya se pueden ver cambios, con ciclos mucho más erráticos, no se encuentran tan marcados los periodos que se podían ver tan claramente en el ciclo anterior. Esto es debido a que, al ser un precio más caro, la batería es más pequeña y se tiende más a "micro ciclos", pequeñas cargas y descargas localizadas durante el día. Estos "micro ciclos" son menos efectivos que las cargas largas en periodos más generales, pero ya que la batería es más pequeña esta solución no sería tan efectiva. No puede permitirse estos largos plazos. Hace lo segundo mejor, con los recursos que tiene, llegando a una solución mucho más compleja.

En la cuarta gráfica se pueden ver mejor estos "**micro ciclos**", se ve un parecido de los tramos generales anteriores pero rodeados de "ruido", subidas y bajadas mucho más esporádicas debido a que al ser mucho más pequeña la batería también se puede cargar y descargar completamente mucho más fácilmente. A tener en cuenta que las baterías suelen dar su vida útil como un número de ciclos. Es decir, puede ser perjudicial para la vida útil de la batería abusar de estos "micro ciclos", se alcanzará este número máximos de ciclos mucho más rápido.

Y como en el caso previo también se puede comprobar la capacidad elegida con esta última gráfica, 1kWh a un 65% de profundidad significa que se tienen 0,65 kWh útiles aproximadamente.

Variaciones de datos

Como ya se ha dicho al inicio de este apartado, es interesante tener variaciones del problema para ver cómo evoluciona la solución. Se harán modificaciones de dos parámetros principalmente, paneles solares a instalar, y demanda. Para el caso de la demanda concretamente, se dispone en el fichero de configuración de un parámetro llamado multiplicador que simplemente multiplica la demanda original por este número. Por defecto está a 1, y no se debería cambiar. A no ser que se busquen otros perfiles teóricos de demandas ("casualmente" el caso a tratar ahora).

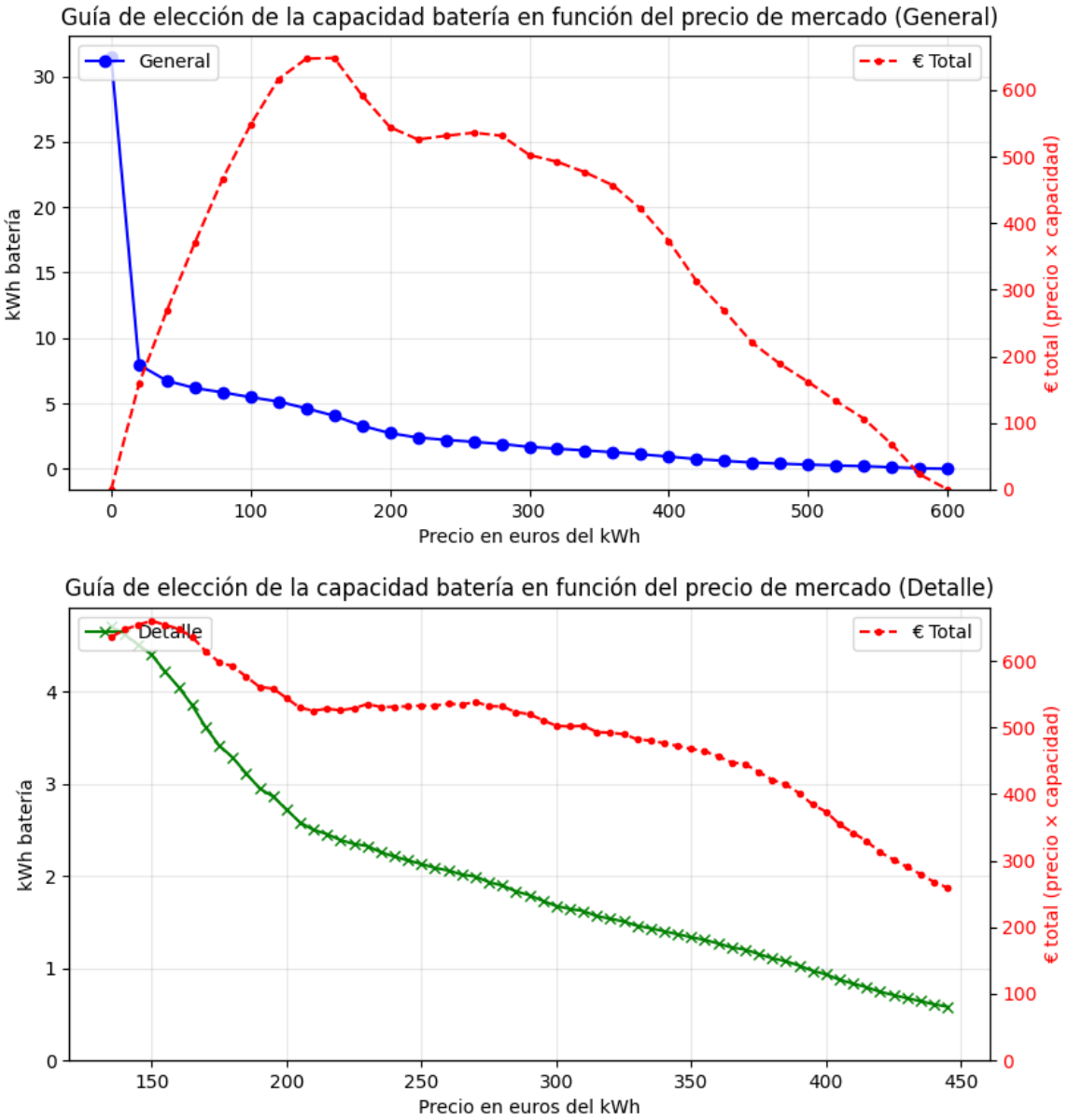


Figura 9. Guía de referencia, multiplicador de demandas x1, 10 m^2 de paneles.

La Figura 9 muestra los mismos datos que la Figura 6 del apartado subapartado anterior, y es la que se usará de base para comparar las permutaciones de configuraciones.

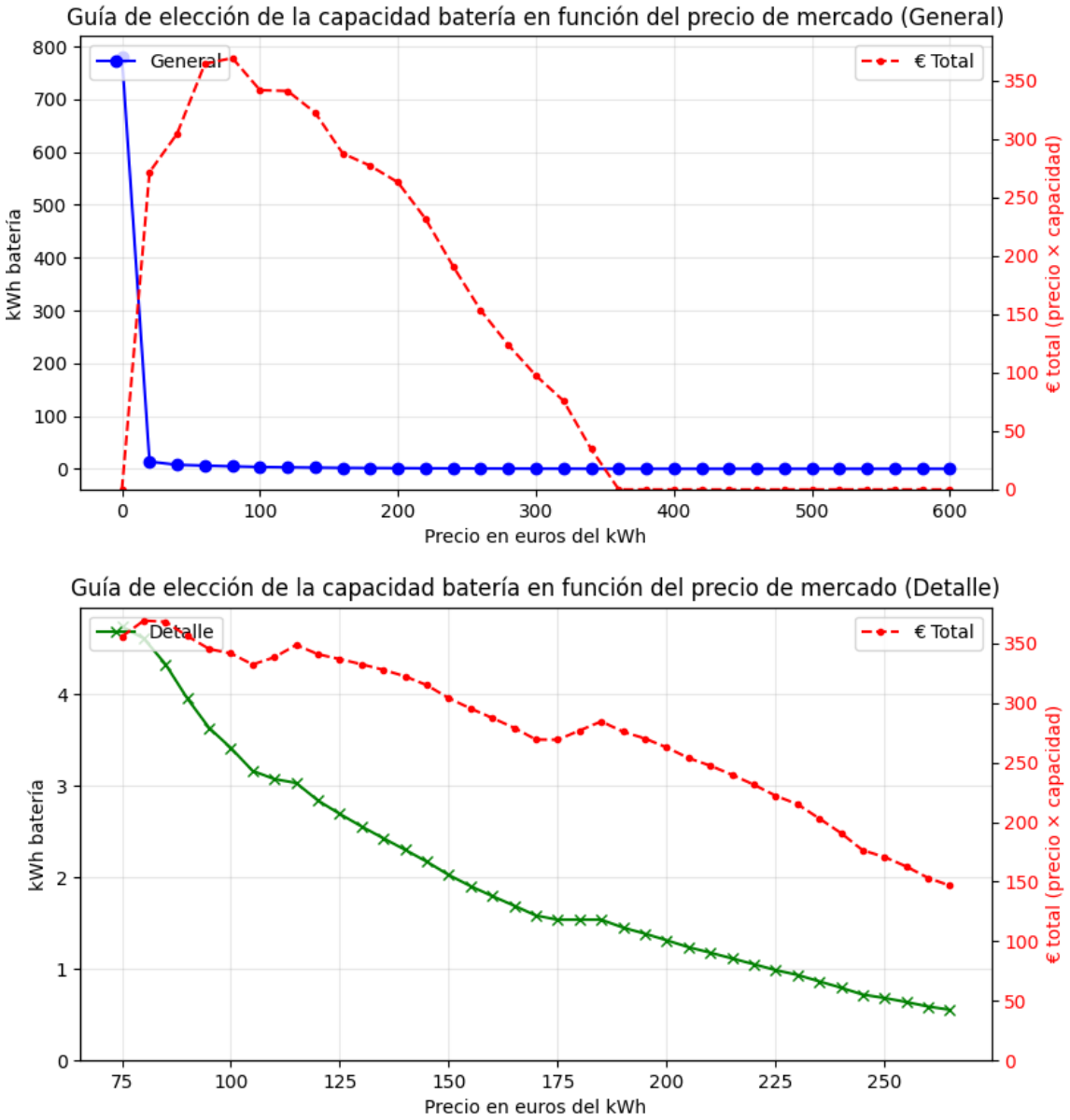


Figura 10. Guía de referencia, multiplicador de demandas x1, 0 m^2 de paneles.

La primera permutación del problema es no instalar paneles respecto a los 10 m^2 de antes. Se puede ver en la Figura 10 cómo varía el problema. El no tener el apoyo de los paneles solares hace que las soluciones viables sean mucho más pequeñas, esta vez el *software* calcula que a partir de poco más de 200 €/kWh apenas hay soluciones viables.

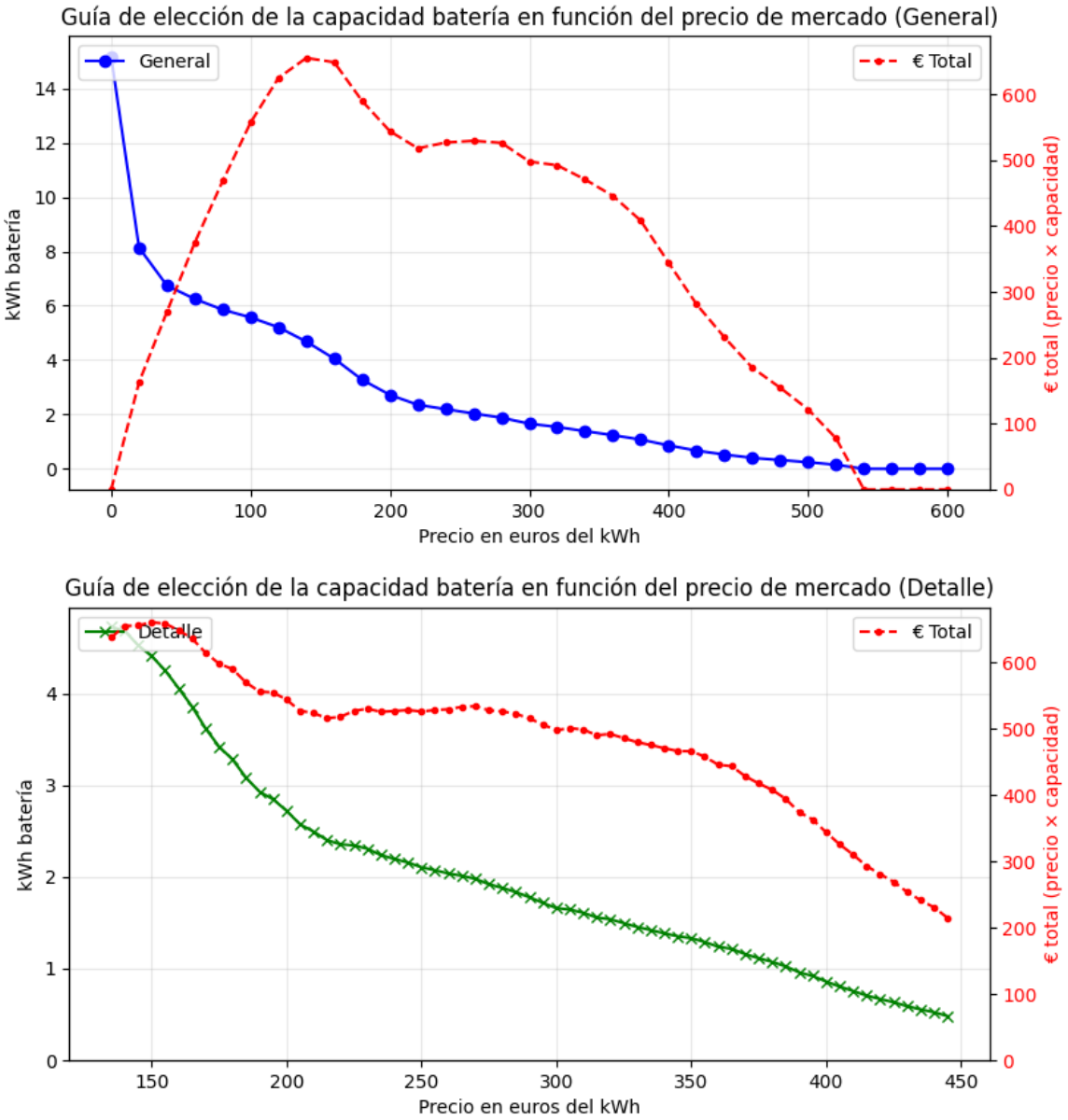


Figura 11. Guía de referencia, multiplicador de demandas x1, 100 m^2 de paneles.

Caso contrario al previo es aumentar hasta valores absurdos los paneles solares. Se podría esperar que basado en la experiencia anterior esta vez el script recomendara valores de batería exageradamente grandes, pero no, la Figura 11 muestra que no hay gran diferencia respecto al caso base de 10 m^2 de paneles solares. Esta poca diferencia se debe a que la demanda no ha aumentado, y por cuestiones de diseño no se permite el volcado de energía a red (venta), así que aun teniendo más paneles el ciclo de batería calculado no es muy distinto.

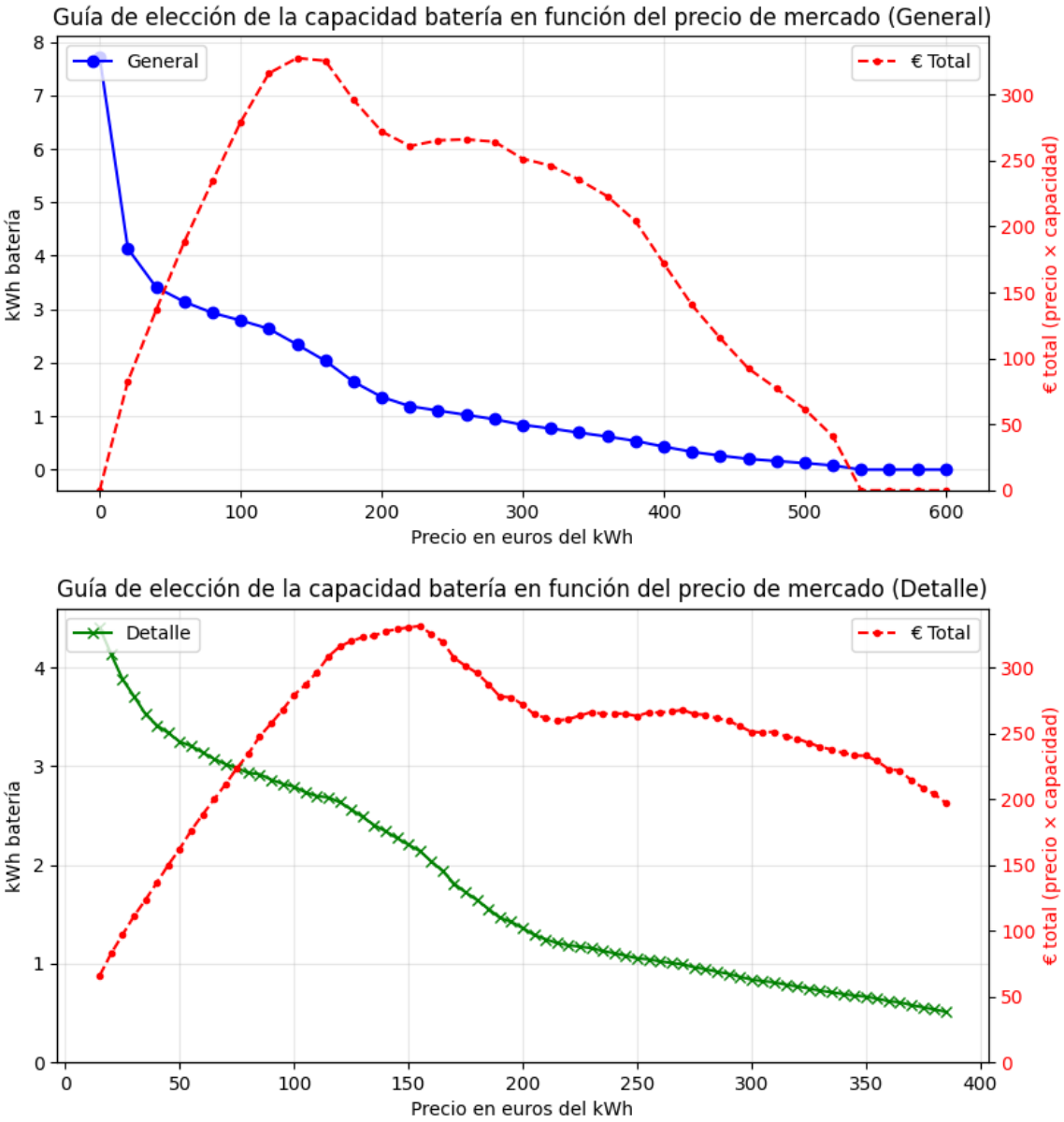


Figura 12. Guía de referencia, multiplicador de demandas x0,5, 10 m^2 de paneles.

Volviendo a los paneles estándares, esta vez se varía la demanda. Con un multiplicador de demanda del 0,5 (la mitad de demanda) se puede ver en la Figura 12 que como se esperaba, si se tiene menos demanda, la batería tendrá menos margen para generar ahorro así que será más difícil justificar baterías más grandes. En el caso base se podían justificar hasta baterías de 400 €/kWh, pero con la mitad de demanda cuesta justificar la instalación con un precio de mercado superior a 300 €/kWh.

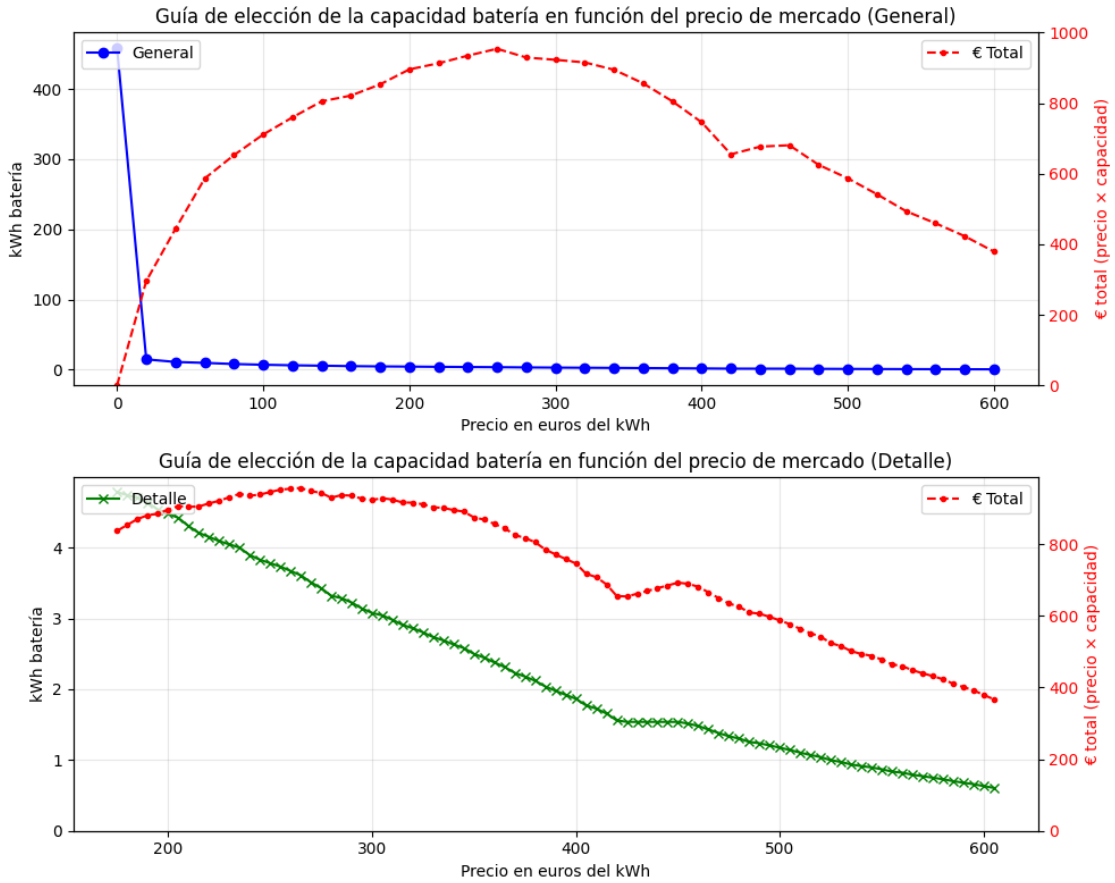


Figura 13. Guía de referencia, multiplicador de demandas x2, 10 m^2 de paneles.

Y repitiendo el patrón, le sigue el caso de aumentar la demanda. Concretamente a duplicarla. Como era de esperar, y sabiendo que la hipótesis inicial era válida, a mayor demanda se puede justificar la instalación de una mayor batería pues hay más margen y opciones de ahorro. Se puede observar la curva de la Figura 13 más extendida, indicando que se podría instalar una batería de hasta 500 €/kWh (un precio extremadamente alto) y aun sería viable la solución.

Resultados del Modo Futuro

Como ya se hizo en el apartado previo, a modo de recordatorio, el **Modo Futuro** tiene como objetivo predecir los datos de próximo día, y en función de estos datos y una elección de una capacidad de batería ya fijada, calcular el ciclo de batería óptimo. El script calcula varios días para asegurar la continuidad entre días del cálculo, pero para efectos prácticos solo será relevante para este apartado un único día, el objetivo. Se usará el modo de visualización de los datos del *software* para tener una información más visual, y además se incluirá alguna información relevante adicional como coste y el ahorro previsto para el día.

Otra cuestión relevante, y como ya se ha mencionado antes es la elección de la batería. Con las gráficas del apartado previo se tiene suficiente información para hacer la elección. Para una comprobación que se hará en posteriores apartados no se usará la guía de los datos “estándar”, pues esa incluye paneles solares (lo recomendado). En su lugar se usará la gráfica que usa la demanda normal pero no usa paneles solares, es decir la Figura 10. Además, ya que estos cálculos incluirán precios es interesante ver la capacidad de ahorro que da la batería sola, sin influencia externa de paneles solares u otras fuentes.

Esta gráfica recomienda muy poca inversión ya que no puede apoyarse en la energía gratis que le proporcionan los paneles, y además la energía consumida es bastante baja. Eso limita las opciones de mercado disponibles, la gráfica apenas recomienda unos 300 o 400 euros de inversión. El mercado local español cuenta con una gran variedad de opciones ya que comparte nicho con la solar, pero debido a la popularidad también provoca que el precio sea ligeramente superior. Esto en un principio no es malo pues este mayor precio es un indicativo de calidad, pero para este caso específico es difícil encontrar una batería a un precio que entre en el presupuesto recomendado. Se debe salir del mercado local entonces. Globalmente, China es el líder en este sector, también en el apartado de precios, pero la batería elegida finalmente para este cálculo fue una de EEUU.

Concretamente la *24V EAGLE* de BatteryEvo (1,5 kWh) (BatteryEVO, 2023). Es una batería de fosfato de hierro y litio (LFP), más barata que una de litio tradicional, pero con menor densidad de energía (más grande físicamente) (Battery University, 2023). Por lo demás tiene los parámetros eléctricos esperados y normales de una batería de litio que se han incluido ya en el cálculo, no es necesario recalcular. Normalmente esta batería costaría 500 dólares, pero se han podido encontrar otros vendedores que la ofertan a 300 dólares (Entropy Survival / BatteryEVO reseller, 2025). Al precio de cambio actual según el ECB a fecha del 9 de septiembre de 2025 (1 EUR = 1,1744 USD (European Central Bank, 2025)), quedaría el suficiente presupuesto dentro de los 300 euros objetivos para pagar el envío. Aún se debería incluir aquí el precio del conversor,

controlador, instalación y otros costes asociados, los cuales deberían ser considerados en una instalación, pero ya que se recomienda hacer esta instalación junto a una solar es difícil evaluar estos precios, se obviarán en este cálculo.

Siguiendo el esquema anterior se incluyen dos apartados, uno de cálculos "estándar", usando los parámetros y datos ya mencionados, y otro apartado con otros cálculos con variaciones del problema, no para este caso específico, pero variaciones del problema. Por la naturaleza de este cálculo y con el objetivo de poder compararlos todos los datos se obtendrán el mismo día, sábado 23/08/2025, para obtener datos del domingo 24/08/2025.

Datos estándar

El cálculo más estándar posible que se puede hacer es el más realista posible. Esto es usando los datos previamente discutidos, batería de 1,5kWh, litio profundidad del 65%, (aún si el fabricante dice que puede ir al 100%), 1 kW de potencia de descarga y descarga (aún si esta puede ir a los 2kW, menos calentamiento alarga la vida útil), sin multiplicador de demanda, y sin paneles solares. Ejecutando los cálculos se obtienen los datos de la Figura 14.

Se puede sacar información de esta Figura 14. Lo primero, las gráficas. Son las mismas que las del modo histórico, pero solo para un día, el día siguiente al cálculo. Estas no muestran datos históricos, pero en su lugar son predicciones. Además, estos serán los datos que recibirá el controlador (junto a algunos más misceláneos y contextuales que no son necesarios en este apartado). El primer dato corresponde a los **datos reales de OMIE** debido a que se ejecutó el programa en una hora avanzada del día (en subapartado de la documentación dedicado a OMIE se explica esto en más detalle), no el dato previsto por la IA. El segundo es el dato de **demanda predicha por la IA**, que puede tener algunos errores típicos de esta herramienta. El tercer dato es el **ciclo de potencia** calculado, el dato que genera el programa. Por último, la cuarta gráfica es la **demanda acumulada de batería**. Se puede ver que empieza a cero, pero podría no hacerlo. Esto es porque el código mantiene continuidad, internamente ha calculado días previos y ha calculado la batería que debería haber quedado del día anterior, usable para este cálculo.

Interpretando los resultados, lo primero que hay que ver es la gráfica de precio de la electricidad. Se ve la bajada de precio que ocurre por la mañana y en torno al medio día. Esto es debido a la época del año en la que se realiza el cálculo (verano) y a la gran cantidad de potencia solar instalada en España, que hacen

que la energía mientras haya sol sea prácticamente o totalmente gratis algunos días (precios de OMIE, los de la distribuidora pueden y serán más elevados). También se puede ver el pico de precio cuando se superan las 19:00 o 20:00.

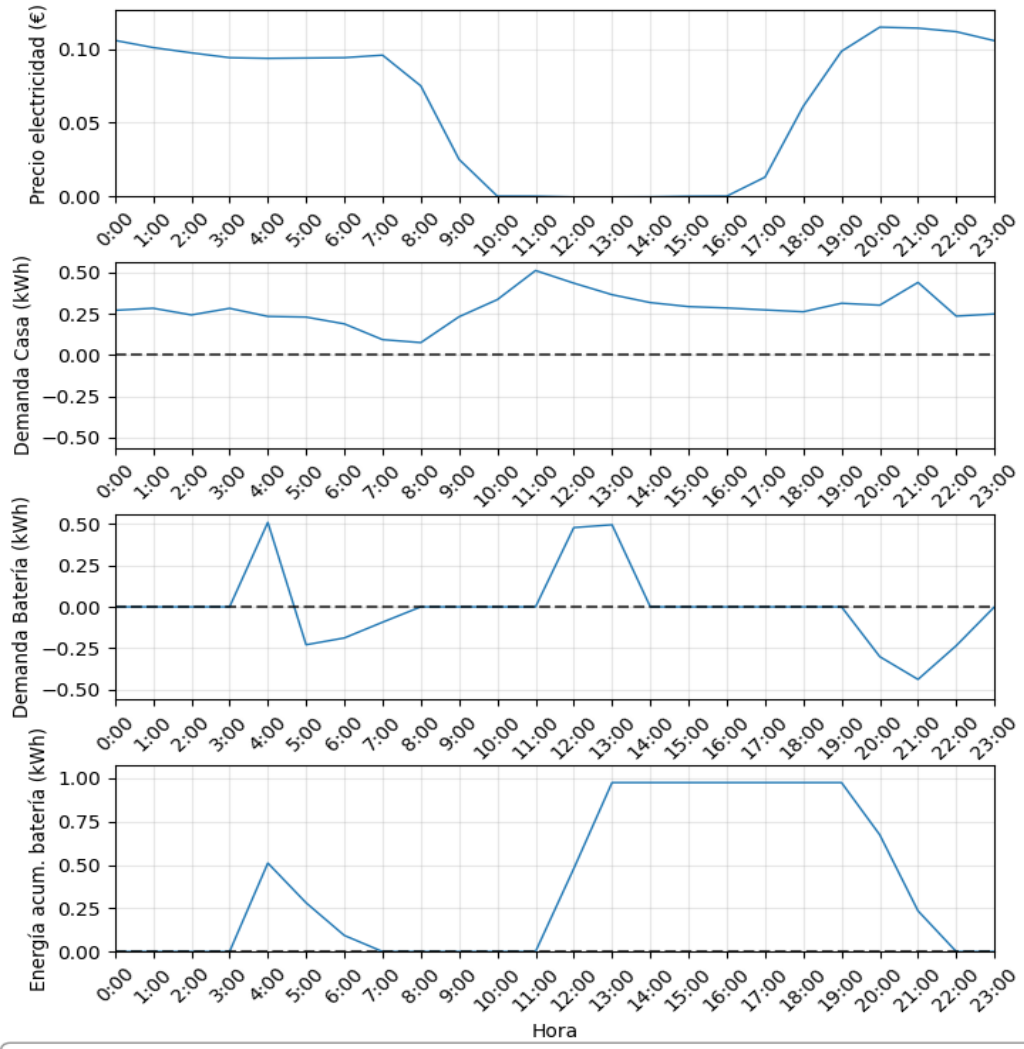


Figura 14.Ciclos predichos para el próximo día, datos estándar(1,5kWh, demanda x1, sin solar).

En cuanto a la demanda, la IA predice que la demanda se mantiene relativamente constante durante el día, salvo dos picos máximos y uno mínimo. El mínimo se da durante la madrugada y los máximos están uno a media mañana (posiblemente la IA ha predicho que el domingo no espera que el usuario empiece a estar activo hasta esa hora), y el otro a la hora pico estándar de la noche.

De este análisis se podría esperar que la batería cargue a las horas más baratas previas a los dos picos, y descargue en ellos. Y la tercera gráfica lo confirma, pero solo uno de los dos. El mayor pico de carga se encuentra a las 11:00, se esperaría un gran ciclo de carga antes de esta hora, pero este pico ya coincide con el tramo de energía prácticamente gratis en esta fecha, por tanto, la batería no necesita actuar. Sí que se ve este gran ciclo de carga y descarga para el segundo pico, el de la noche. Se observa una carga a mediodía y una descarga en este pico. De esta tercera gráfica se infiere la potencia máxima de 1kW también. La cuarta gráfica da una información parecida, se puede ver una pequeña carga y descarga durante la noche, y la mayor parte de la gráfica que carga durante la tarde y la mantiene hasta la noche. También de la cuarta gráfica se puede sacar la potencia máxima usable, cercana a 1kWh (lo cual tiene sentido porque la máxima es 1kWh, pero solo usable un 65%, quedan disponibles 0,975 kWh).

Finalmente, se incluye un pequeño **cuadro con precios**. Su descripción ya se puede intuir qué información darán, pero para dar más detalle, el dato arriba a la izquierda, el coste sin batería, es simplemente multiplicar el precio de la electricidad hora a hora por la suma de demanda consumida más la potencia generada por los paneles, 0 en este caso. Este es el precio que se hubiera pagado normalmente, sin la instalación de este sistema. El dato de arriba a la derecha es similar, pero se suma a esta pequeña ecuación la demanda de la batería, y este será el coste a optimizar. Este es el precio que se pagaría si se hubiera instalado la batería. El dato de abajo a la izquierda es el ahorro que habría generado el sistema de la batería. Para este día concreto el sistema pudo pasar de los 38 céntimos sin batería, a los 26 con batería, dando un ahorro de 11 céntimos, es decir un ahorro del 29%. Sin embargo, este dato no incluye la inversión inicial en la batería. Si se quiere ver cómo quedaría este ahorro incluyendo la amortización diaria del sistema se debe ir al último dato de este cuadro, el de abajo a la derecha, el cual sí incluye esta información. Restando esta inversión (viendo la diferencia entre ambos se puede ver que el programa calculó 8 céntimos diarios, es decir 300 euros entre 10 años a 365 días) el sistema solo calcula un ahorro de 3 céntimos, es decir un ahorro del 8%.

Como se esperaba, para tan poca demanda el coste del sistema era el factor más limitante. Se puede ver que funciona bastante bien de por sí, casi un 30% de ahorro, pero cuando se descuenta el precio a invertir el ahorro si bien no está mal, un 8% es un margen respetable, siguen siendo solo 3 céntimos de ahorro.

Variaciones de datos

Viendo los datos obtenidos en el subapartado previo la primera pregunta que nace es, ¿cómo afectaría a los resultados una capacidad de batería distinta? También puede ser interesante usar la herramienta del multiplicador de demanda para obtener más variaciones del problema como se hizo previamente.

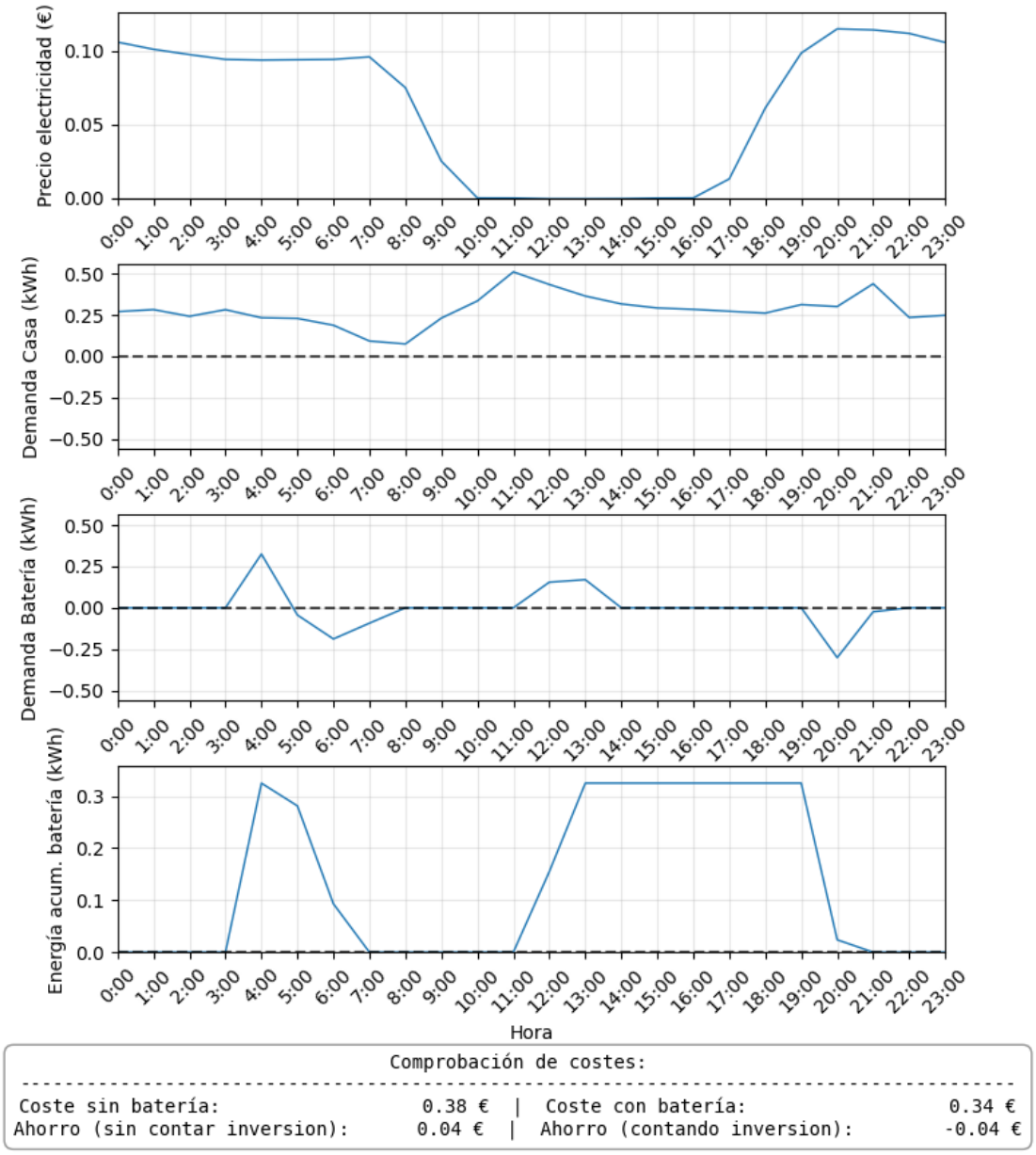
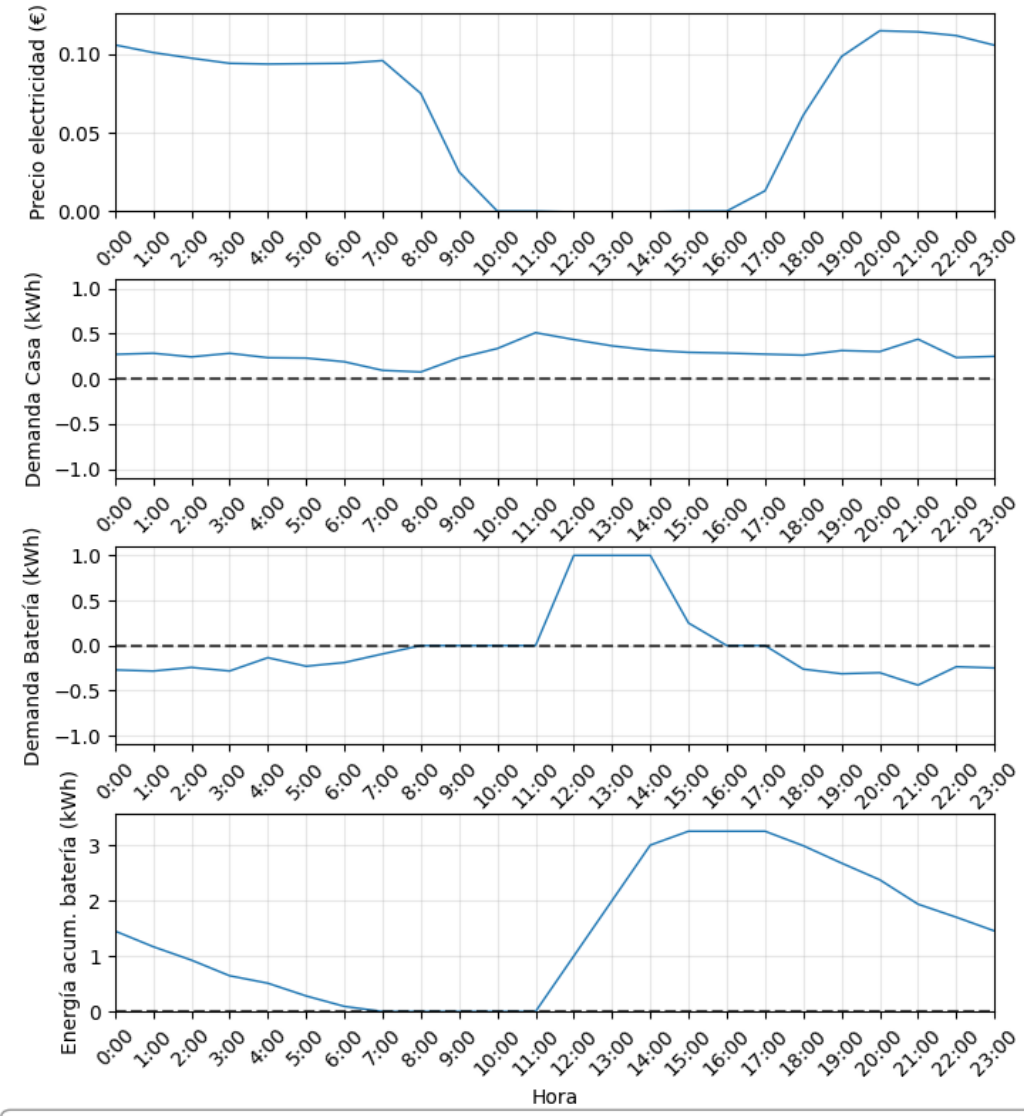


Figura 15. Mismos parámetros (multiplicador x1, sin paneles), pero con una batería de 0,5 kWh.

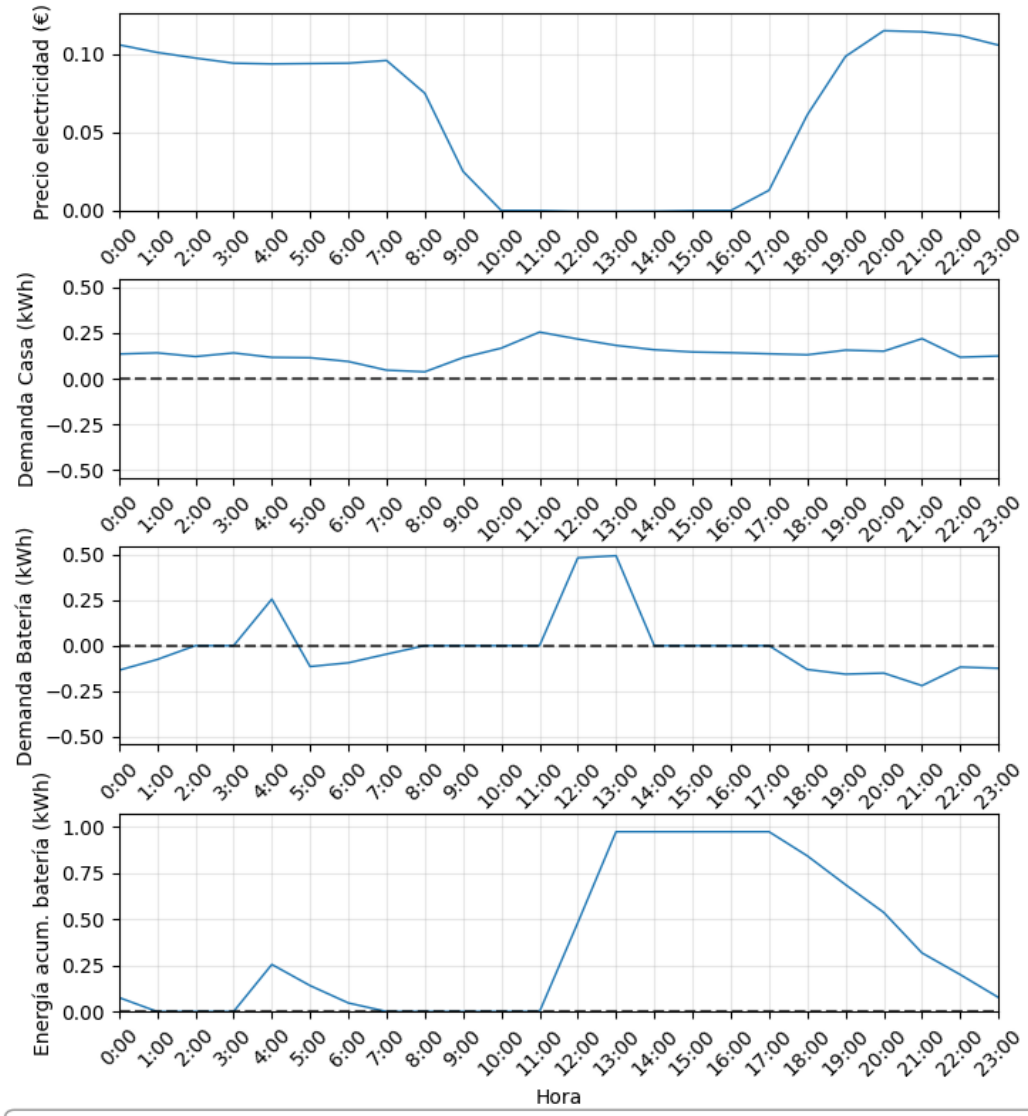
En línea con lo esperado, se observa en la Figura 15 que cuando se reduce la capacidad de la batería a 1/3, pero manteniendo el coste, se está incrementando en esencia el precio del kWh por tres, lo cual estaba altamente desaconsejado por la gráfica de guía del subapartado anterior (Figura 10). La topología de los ciclos y las soluciones elegidas en general se mantienen, pero la batería es simplemente demasiado cara para tan poca capacidad para ser justificable. Una vez se incluye la inversión necesaria se pierde dinero (ahorro negativo).




```
                    Comprobación de costes:
-------------------------------------------------------------------------------------
Coste sin batería:                0.38 €  |  Coste con batería:                0.02 €
Ahorro (sin contar inversion):    0.35 €  |  Ahorro (contando inversion):      0.27 €
```

Figura 16. Mismos parámetros (multiplicador x1, sin paneles), pero con una batería de 5 kWh.

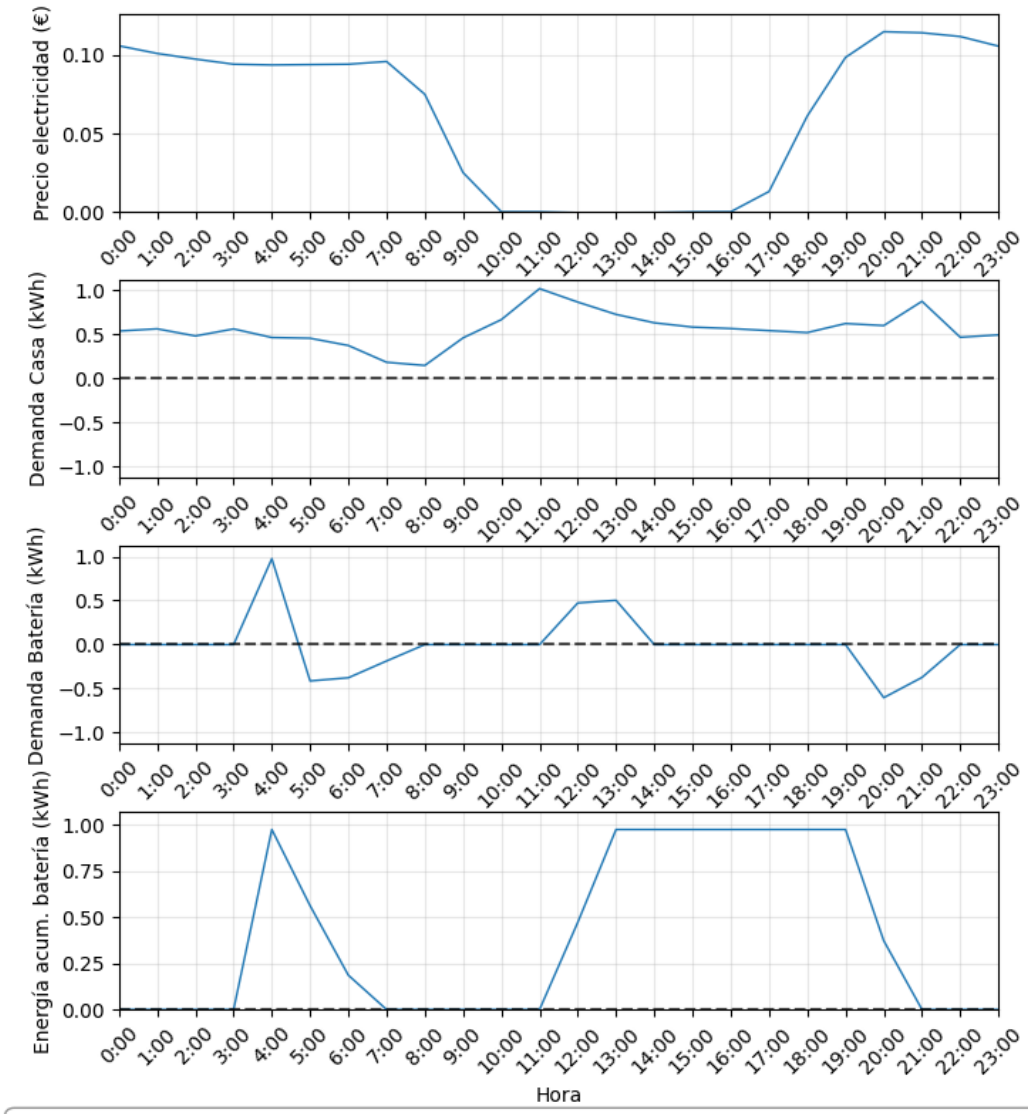
Contrario al caso anterior, la Figura 16 muestra cómo ahora se ha aumentado la capacidad más de tres veces, pero manteniendo el coste constante. La gráfica de guía (Figura 10) recomienda cómodamente estos valores de batería. Y no hay mejor justificación que ver los 27 céntimos ahorradas en el día, un ahorro de más del 70% aun contando la inversión. La batería es tan grande que puede permitirse cargar casi toda la energía del día a las horas en las que es prácticamente gratis. Pero este tamaño y precio simplemente no son factibles, no es posible encontrar una batería de 5kWh de calidad aceptable por solo 300 euros.




```
                    Comprobación de costes:
-------------------------------------------------------------------------------
Coste sin batería:              0.19 €  |  Coste con batería:              0.07 €
Ahorro (sin contar inversion):  0.11 €  |  Ahorro (contando inversion):    0.03 €
```

Figura 17. Mismos parámetros (batería de 1.5kWh, sin paneles), pero con multiplicador de x0,5.

En la Figura 17 se ve que en este caso se ha reducido la escala de la demanda consumida, pero manteniendo su perfil. La solución elegida sigue la misma línea que la original. Pasa similar al caso anterior, la demanda es tan baja que aún con una batería relativamente pequeña como esta, puede aliviar grandes costes de la demanda. Como puede verse, ahorra casi el 60% del precio sin contar la inversión. Pero la inversión es el problema aquí, al costar la batería lo mismo, el ahorro que puede obtener neto es igual al original. Se está en el límite de la batería, aún es usable para esta demanda, pero no es más eficiente que para su demanda calculada originalmente (aunque tampoco menos).




```
                    Comprobación de costes:
-------------------------------------------------------------------------------
Coste sin batería:              0.75 €  |  Coste con batería:              0.64 €
Ahorro (sin contar inversion):  0.11 €  |  Ahorro (contando inversion):    0.03 €
```

Figura 18. Mismos parámetros (batería de 1,5kWh, sin paneles), pero con multiplicador de x2.

Caso curioso respecto al anterior, la Figura 18 muestra que si se aumenta la demanda el precio ahorrado vuelve a ser el mismo. Esta vez se ha llegado al límite, pero “por el otro lado”, es decir, la demanda es tan grande que se puede ver en los costes brutos que la batería no es lo suficientemente grande como para tener un impacto real en la factura. Se debería recalcular una guía para estas demandas y elegir en función de esta. En este caso hay margen de ahorro disponible aún.

Comprobación del software.

Una vez se tienen los resultados que genera el script queda una pregunta por resolver, **¿funciona realmente?**

Esa pregunta solo se puede resolver instalando el sistema y dejándolo funcionar realmente y obtener de ahí datos empíricos con el tiempo. Sin embargo, si bien están fuera del alcance de este proyecto los datos reales, sí que se pueden obtener los datos teóricos en ese mismo tiempo, y comparar los datos que hubiera calculado el *software* usando la IA con los datos reales. Ya que ha pasado tiempo desde que se descargó el histórico con el que se ejecutaron los datos, se pueden usar los datos históricos generados naturalmente durante ese tiempo como datos reales, mientras que los datos generados por la IA serían los que serían objeto de estudio.

El *software* al calcular el día siguiente genera un registro de datos que no se han usado para el cálculo, pero que han sido generados por el *software*. Los datos que influyen en el cálculo solo serán los de demanda del usuario, los precios de electricidad y los de potencia solar. En cuanto a los paneles solares ya que el sistema no fue instalado, similar al apartado previo, simplemente se pueden obviar, se ponen a 0. Para precios, mientras el *software* se ejecute en las últimas horas del día, el programa usará precios de electricidad reales. De modo que este dato tampoco será un problema. Solo resta la demanda. Simplemente se han de tomar datos reales de la web de eDistribución, y hacer un cálculo por separado. Uno usando los datos que se tienen ahora, siendo calculados como predicciones con la IA, y el otro usando como input los nuevos datos históricos que se han generados con el pasar del tiempo natural. De este modo se tendrá una comparación entre los datos reales y los datos predichos con la IA, y ver cómo de efectiva sería la IA y el sistema.

Es decir, el precio normalmente se calcula haciendo sumatorio de demandas (usuario, batería, solar) y multiplicándolo por el precio de la electricidad. Para hacer la comparación se sumará el ciclo de demanda de la batería calculado fruto de predicciones con IA en la demanda real (sin IA). Esta suma multiplicada por el precio de la electricidad debería ser más bajo que el coste de la demanda real, pero no tan bajo como el de la demanda predicha (se calculó un ciclo óptimo para esta demanda, no será posible superarlo).

Este cálculo es no algo que esté implementado en el código estándar, ya que es una comprobación externa. Sin embargo, una vez se tienen los datos es un cálculo muy sencillo, y no son necesarias gráficas extras, solo se necesitan resultados numéricos normales. Se puede implementar externamente sin mayores problemas, incluso usando programas como Excel.

Se adjunta dicho Excel en la propia carpeta de outputs del *software*, en una carpeta auxiliar para esta memoria, junto con los datos fuente usados. Si bien esta carpeta está dedicada a los datos que el *software* genera continua y autónomamente, y realmente este tipo de datos no tienen cabida en esta carpeta, se considera más conveniente juntar todos los datos de resultados.

El Excel toma los datos y cálculos de 4 escenarios, precios sin baterías, tanto usando demandas reales como predichas por la IA, así como precios con la batería, nuevamente reales y predichas. Multiplica valor a valor y luego hace un sumatorio. Estos sumatorios se incluyen en la Tabla 1, y a partir de ellos se derivan el resto de cálculos y conclusiones. Para empezar, se puede ver que la IA tiende a predicciones de demandas más bajas que las reales. El consumo real es aproximadamente un tercio más alto que el predicho. Esto es indicativo de que o bien los patrones de consumo han cambiado o que el modelo de IA no es lo suficientemente bueno. Es difícil saber la respuesta sin un análisis en más profundidad, pero es más posible que sea un defecto de IA que un cambio de patrón. Se debería investigar y mejorar. Lo segundo que se puede ver es que como ya se esperaba, es que el ciclo de demanda es mucho más efectivo en el ciclo para el que fue calculado (predicho) que para el real. El coste con la batería del predicho es casi la mitad del real. Esta diferencia tan alta también es en parte debida a cómo se calcula este precio en el Excel, ya que si la suma de demandas es negativa entonces se pasa a 0. Esta condición va en la línea del no verter a la red, pero sin un algoritmo de control que regule mejor este ciclo, simplemente se pierde y desperdicia no solo una gran capacidad útil de la batería, sino además la propia energía. Una buena parte de esta se pierde cuando se hace la comprobación de si el sumatorio de demandas es menor que 0, entonces se hace 0 el total. Esta condición elimina completamente energía, haciéndolo altamente ineficiente, perdiendo más de la mitad del ahorro en el proceso.

Por último, queda ver el ahorro bruto, tanto el predicho como el real. Se ve que el predicho puede ahorrar 12 euros en 150 días, mientras que el real solo podría haber ahorrado con este algoritmo tan burdo de adaptarse a la realidad poco más de 7 euros. Esto equivale a apenas 8 y 5 céntimos respectivamente, lo cual es bastante poco, pero es lo que se puede obtener de estas demandas tan bajas. Sin embargo, ya que estas demandas son bajas, el porcentaje ahorrado en comparación a lo gastado es bastante bueno, 43% y 19% respectivamente. Nótese que estos precios tan bajos son fruto de usar los precios de OMIE, que serán mucho menores a los de la distribuidora final.

Se puede tomar este caso como uno de los casos factibles con menor potencia consumida, así que es interesante caracterizar este caso para tenerlo como referencia. Lo más inmediato es tomar el dato de la Tabla 1 del sumatorio de potencia real consumida, 821,33 kWh. Esto es durante un periodo de 150 días, aproximadamente 5 meses. Por tanto, la **demanda media mensual es de 164,27 kWh** (821,33 / 5 = 164,27).

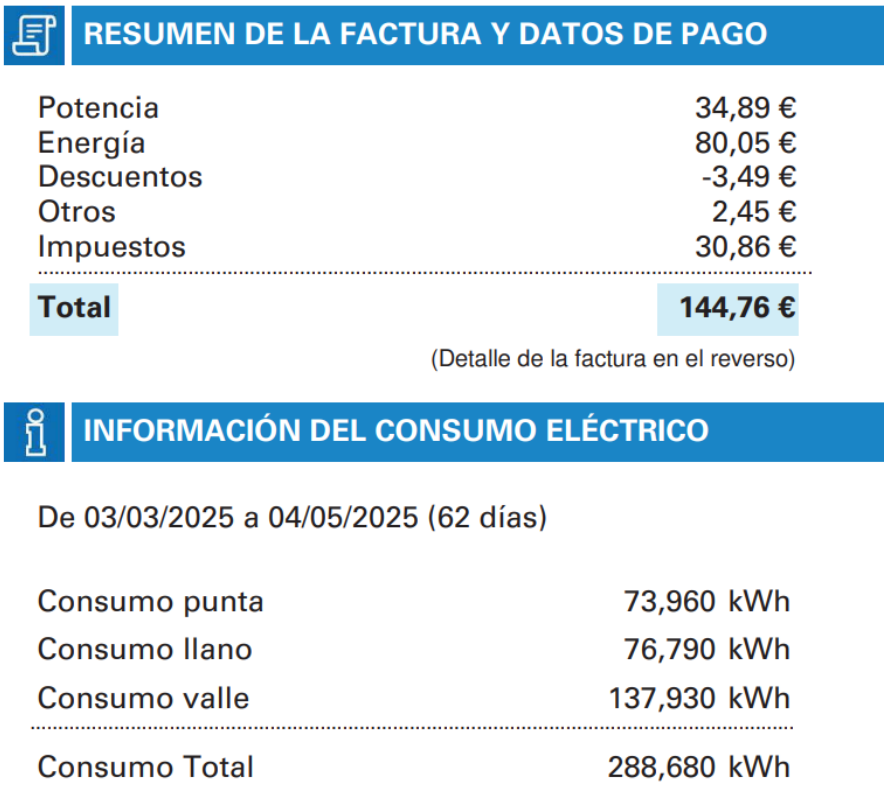
Días calculados	->	150
RESULTADOS		
Energía consumida total predicha	->	654,05 kWh
Energía consumida total real	->	821,33 kWh
Sumatorio de precios sin batería (datos predichos)	->	28,22 €
Sumatorio de precios sin batería (datos reales)	->	37,44 €
Sumatorio de precios con batería (datos predichos)	->	16,17 €
Sumatorio de precios con batería (datos reales)	->	30,26 €
Ahorro con precios OMIE (predicho)	**->**	**12,05 €**
Ahorro con precios OMIE (real)	**->**	**7,18 €**
Ahorro % (predicho)	**->**	**43%**
Ahorro % (real)	**->**	**19%**

Tabla 1. Resultados comparando datos reales con predichos durante 5 meses.

También es interesante comparar los datos “brutos” de energía aquí calculados a precios de distribuidora y compararlos con los reales de Endesa. La facturación de Endesa es bimensual, así que para este periodo de 150 días existirán dos facturas. No se adjuntarán completas por cuestiones de privacidad, pero sí se referenciarán datos relevantes.

La primera factura de este rango es la del 03/03/2025 al 04/05/2025, que si bien empieza unos días antes del rango a calcular es lo mejor que se tiene, y es suficiente para el objetivo de esta comparación. La Figura 19 indica que para este periodo de aproximadamente 2 meses se tiene un gasto de 144,76 €, de los cuales 80,05 € son exclusivos de la energía. Esto es aproximadamente 40,03 € al mes (80,05 / 2 = 40,025). Lo cual es significativamente superior a lo calculado en la Tabla 1 a precio de costo de OMIE, siendo el coste total real para los 150 días de cálculo (5 meses) sin baterías ni software aplicado de 37,44 €. Es decir, **el precio de la energía por parte de Endesa es 5 veces superior al precio de costo de OMIE.**

Se puede descartar que se dé un consumo superior al previsto, pues la Figura 19 también indica que se consumieron 288,680 kWh, lo que lo deja en una media mensual de 144,340 kWh (288,680 / 2 = 144,340), en la línea de la aproximación de los 164,27 kWh mensuales estimados del documento de cálculo. La razón de este incremento de coste se ve en la Figura 20, la cual desglosa el precio de la energía. Se puede ver que incluso las horas más baratas, las valle, las cuales OMIE las ofertaba prácticamente gratis por la abundante energía solar, Endesa las cobra a más de 0,24 €/kWh. Precio incluso superior a la hora más cara de OMIE del documento de cálculo (0,21 €/kWh).



RESUMEN DE LA FACTURA Y DATOS DE PAGO

Potencia	34,89 €
Energía	80,05 €
Descuentos	-3,49 €
Otros	2,45 €
Impuestos	30,86 €
Total	**144,76 €**

(Detalle de la factura en el reverso)

INFORMACIÓN DEL CONSUMO ELÉCTRICO

De 03/03/2025 a 04/05/2025 (62 días)

Consumo punta	73,960 kWh
Consumo llano	76,790 kWh
Consumo valle	137,930 kWh
Consumo Total	288,680 kWh

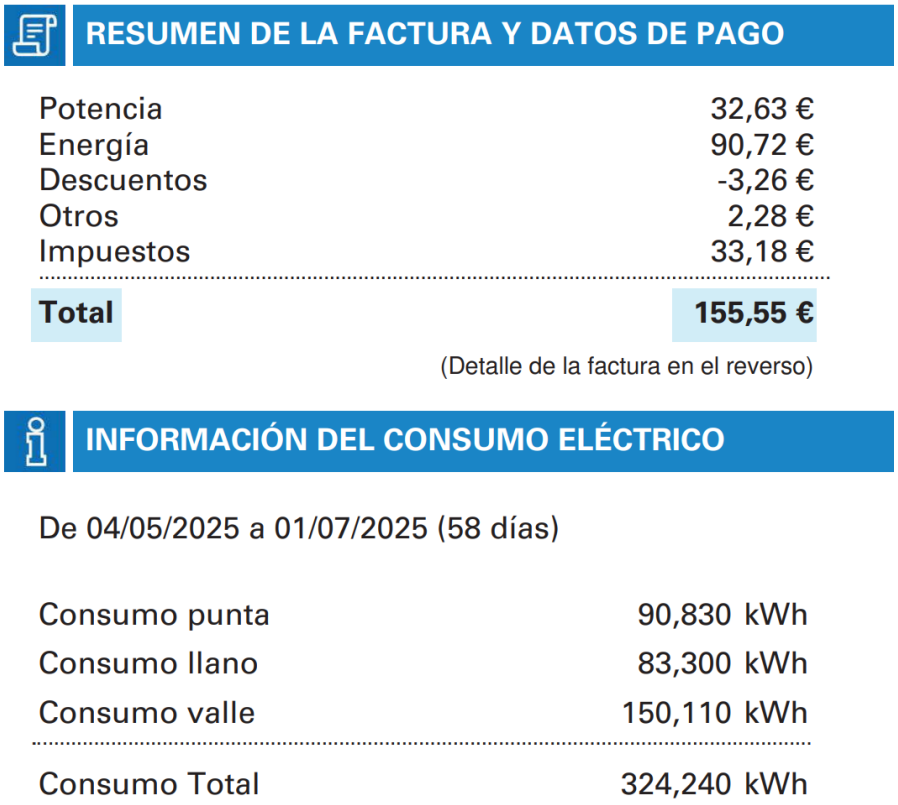
Figura 19. Factura de Endesa del 03/03/2025 al 04/05/2025, precio total.

Energía	**80,05 €**
Consumo Punta 40,870 kWh x 0,348178 Eur/kWh	14,23 €
Consumo Punta 33,090 kWh x 0,348639 Eur/kWh	11,54 €
Consumo Llano 37,160 kWh x 0,271042 Eur/kWh	10,07 €
Consumo Llano 39,630 kWh x 0,271503 Eur/kWh	10,76 €
Consumo Valle 59,450 kWh x 0,242254 Eur/kWh	14,40 €
Consumo Valle 78,480 kWh x 0,242715 Eur/kWh	19,05 €

Figura 19. Factura de Endesa del 03/03/2025 al 04/05/2025, desglose del precio de la energía.

Se obtienen datos similares de la factura del 04/05/2025 al 01/07/2025, esta vez con datos con fechas completamente en el rango del documento de cálculo. Esta vez la Figura 21 indica que se pagan 155,55 €, de los cuales 90,72 € son exclusivos de la energía, lo que sube la media de precio por la energía mensual a los 45,36 €. Y en cuanto a la parte energética esta vez fueron 324,24 kWh, dejando la media mensual en 162,12 kWh, perfectamente en rango con lo calculado y esperado.

Similar pasa con la Figura 22, con los precios más bajos de Endesa, los valle, superando los máximos de OMIE.



RESUMEN DE LA FACTURA Y DATOS DE PAGO

Potencia	32,63 €
Energía	90,72 €
Descuentos	-3,26 €
Otros	2,28 €
Impuestos	33,18 €
Total	**155,55 €**

(Detalle de la factura en el reverso)

INFORMACIÓN DEL CONSUMO ELÉCTRICO

De 04/05/2025 a 01/07/2025 (58 días)

Consumo punta	90,830 kWh
Consumo llano	83,300 kWh
Consumo valle	150,110 kWh
Consumo Total	324,240 kWh

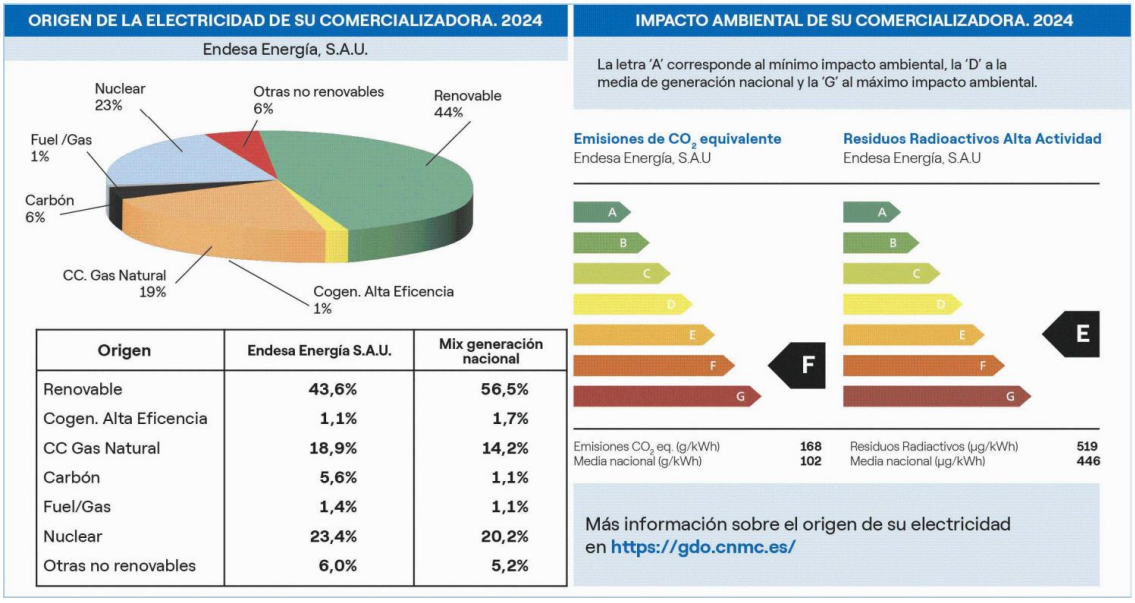
Figura 20. Factura de Endesa del 04/05/2025 al 01/07/2025, precio total.

Energía ..... **90,72 €**
Consumo Punta 90,830 kWh x 0,348639 Eur/kWh ..... 31,67 €
Consumo Llano 83,300 kWh x 0,271503 Eur/kWh ..... 22,62 €
Consumo Valle 150,110 kWh x 0,242715 Eur/kWh ..... 36,43 €

Figura 21. Factura de Endesa del 04/05/2025 al 01/07/2025, desglose del precio de la energía.

Con estos datos se llega a varias conclusiones. La primera es que como se esperaba, los datos de demanda son correctos. La segunda es que todo el cálculo se ha hecho basado en los precios de OMIE, los cuales, si bien se espera que la forma y las tendencias reflejen a los de las comercializadoras, son mucho más bajos que los de estas. Y tercero, que el contrato actual con Endesa podría funcionar, pues separa consumos en franjas horarias a distintos precios, justo lo que necesita el software desarrollado. Pero a su vez también se podría mejorar con uno que se apegue mejor a los precios de OMIE, no solo en escala, sino también en forma.

Como apunte final, si bien no directamente relacionado con este trabajo, la última hoja de la factura de Endesa puede aportar información del motivo de estos precios tan inflados, Figura 23. Endesa usa mucho menos energía renovable, la más barata, que la media nacional. Energía que sustituye por gas natural y carbón, la más cara. Esto no solo lleva a precios mucho más caros, sino que además a unas emisiones de CO_2 mucho más altas que la media (la media es D, mientras que Endesa es F).



Origen	Endesa Energía S.A.U.	Mix generación nacional
Renovable	43,6%	56,5%
Cogen. Alta Eficencia	1,1%	1,7%
CC Gas Natural	18,9%	14,2%
Carbón	5,6%	1,1%
Fuel/Gas	1,4%	1,1%
Nuclear	23,4%	20,2%
Otras no renovables	6,0%	5,2%

Figura 22. Factura de Endesa del 03/03/2025 al 04/05/2025, mix energético y eficiencia.

Pero con esta información se buscaba caracterizar estos datos para tomarlos como caso base para usar este sistema. Se puede mantener el dato del **consumo energético medio de 164,27 kWh/mes**, pues se ha demostrado que está en la línea de lo consumido según informan las facturas.

Y en cuanto a precios de factura, la primera es de 144,76 € y la segunda de 155,55 €, ambas bimensuales. Por tanto, la media bimensual de ambas sería de 150,15 € ((144,76 + 155,55) / 2), o lo que es lo mismo, un **coste medio de 75,08 €/mes** (150,15 / 2 = 75,075).

Por tanto, redondeando y resumiendo, y siempre teniendo en cuenta que el patrón de consumo será relevante y por tanto esta aproximación no podrá sustituir apropiadamente un buen estudio con el cálculo expuesto durante esta memoria, se puede tomar **este caso como el rango inferior de aplicabilidad del software, con un consumo de aproximadamente 165 kWh/mes, a un precio de 75 €/mes.**

Comparación con la realidad

Como última verificación, se puede comprobar qué soluciones se están instalando en la realidad, sobre todo en el sector solar, pues es la comparación más directa que se tiene. Como se ha remarcado varias veces, si bien es un concepto distinto, comparte tecnología con este sector, y por tanto es de esperar que comparta soluciones.

Una rápida búsqueda por internet para ver qué tipo de productos se ofrecen para autogeneración con solar doméstica lleva a la web AutoSolar. Esta web está especializada en la instalación completa, incluyendo también los propios paneles solares, así como la electrónica asociada. La información más relevante que se puede sacar de aquí para comparar resultados con los resultados obtenidos es el tipo de baterías usadas y la capacidad de estas.

Se puede ver un amplio rango de precios y gran variedad de opciones, pero ordenando por relevancia, la primera opción es el kit *Azzurro.* Este kit se presenta con una batería de litio 5 kWh, aunque la realidad es que al bajar y buscar más información de la batería tiene 4,8 kWh, y con una profundidad de ciclo del 95% la deja en poco más de 4,6 kWh útiles (4,80 kWh * 0,95 = 4,56 kWh) (AutoSolar, 2025). A actualmente 1235 €, su precio por kWh queda a poco más de 270 €/kWh (1.235 € / 4,56 kWh = 270,83 €/kWh) (AutoSolar, 2025). Se puede ver por tanto que esta batería estaría en la horquilla más amplia que se determina más útil para este problema, pero por lo demás, el resto de datos asumidos y obtenidos se ve que son perfectamente razonables. Capacidad en rango esperado, ciclo recomendado mucho más favorable al calculado, misma tecnología, y precio unitario del kWh perfectamente en el rango calculado. Aunque para los datos de demanda que se tienen, esta batería sería demasiado grande, la demanda no es la suficiente, por eso se decidió usar una batería de solo 1 kWh.

Si se busca algo más parecido a la solución elegida y calculada en este capítulo, se debe bajar un poco en la lista ordenada por relevancia dentro de la misma web. Y si bien se encuentran soluciones, al ser una batería tan pequeña la que se ha decidido usar por ni usar paneles solares ni tener un gran consumo, las opciones disponibles bajan. Son de hecho las soluciones portátiles las más comunes, no son tan representativa de este escenario.

Se puede comprobar entonces que el sistema genera soluciones alineadas con el precio de mercado, pero el caso específico calculado se dan unas demandas tan pequeñas que la solución empieza a ser demasiado específica como para acomodarse al mercado común. Pero también se puede comprobar que los **parámetros y la configuración elegidos son correctos**, y que el sistema genera **soluciones coherentes**.

TRABAJO A FUTURO

Vistos los resultados los resultados son optimistas. Pero también revelan algo importante. La Tabla 1 hace una comparación de la realidad con el modelo teórico, en el cual se pierde un 24% del ahorro. He aquí la continuación de este trabajo, la correcta implementación física de este *software.*

Como se dice en el subapartado anterior donde se presenta dicha Tabla 1, la comprobación del *software*, esta suma es muy simple, pues en ambos se hace un sumatorio de demandas y lo multiplica por el precio. En el caso de que dicha suma sea negativa (la batería descarga sin que haya tanta demanda) entonces hará el sumatorio 0, desperdiciándose en esencia parte de la energía. En el modo ideal esta condición no es un problema, pues está perfectamente ajustado para que esto no ocurra, con una precisión matemática que solo podría ser fruto de unas condiciones teóricas. Pero en la realidad esto no ocurre así. La demanda del usuario puede y va a ser distinta de la prevista, es así la naturaleza de este problema, lleva implícita una aleatoriedad que nunca se podrá predecir perfectamente.

Esta es la aproximación más burda que se podría aplicar al problema como ya se puede suponer, no solo ineficiente pero además en contra de los preceptos del propio *software*, controlar la demanda y prevenir vertidos a red indeseados. Pero saca a relucir la importancia de la implementación física. El software desarrollado genera un ciclo previsto, y si la demanda del usuario es exactamente igual a la prevista entonces el controlador de la batería no deberá hacer nada especial, simplemente apegarse a la guía. Pero este no será el caso, la demanda no será igual, y entonces será el trabajo del controlador real el que deba decidir a tiempo real qué hacer.

En previsión a esta tarea, el programa no solo manda al controlador el ciclo de batería, además manda los datos en los que se basó para tomar esa decisión, como puede ser la demanda del usuario prevista o el precio de la electricidad, así como otros datos relevantes. Por tanto, la implementación física del *software* deberá tener un algoritmo que en función de la información que le llega de este software sea capaz de adaptar estos datos teóricos a la realidad. Adaptar una solución teórica al medio físico.

Será entonces esta correcta implementación del controlador y su algoritmo la que juegue un gran papel en la eficiencia final de este programa desarrollado. Nuevamente, en la Tabla 1 se ve un ejemplo de una mala implementación de

este algoritmo, su eficacia se reduce a menos de la mitad. Ahora bien, ¿qué formas podría tener este algoritmo?

Primero se ha de tener en mente que este controlador tendrá un hardware poco potente, así que no se podrá ejecutar un cálculo ni remotamente similar al que se hace en este software. Pero tampoco debería ser necesario pues este cálculo será uno de los datos de entrada, y si bien la demanda predicha por la IA no será exactamente igual, será lo suficientemente parecida. El controlador tendrá también a su disposición datos de demanda prevista, precios, temperatura y otros parámetros relevantes, pero debidamente dosificados y fáciles de procesar, o si no se corre el riesgo de desbordar la capacidad de cálculo de la plataforma elegida. A su vez, al estar físicamente implementado en la red del usuario, este controlador debería tener acceso a la demanda en tiempo real de dicho usuario, siendo este el último dato necesario para poder ejecutar este control.

Con esta información ya se podrá desarrollar el algoritmo más adecuado. Un ejemplo de dicho algoritmo podría ser simplemente seguir el nivel de carga porcentual calculado teóricamente, y en el caso de que se quiera hacer una inyección de potencia superior a la demanda en ese punto, simplemente no hacerla, y redistribuir esa potencia sobrante en un punto de demanda a un precio caro. Pero aparecen cuestiones, tales como ¿qué hacer si en las horas más caras del día hay más demanda de la prevista?, ¿se debería seguir sin más la previsión y ahorrar carga para unas horas posteriores, previendo más demanda a una hora aún más cara, o será este el pico de demanda del día, que simplemente se dio a una hora más temprana y no tendría sentido ahorrar batería? O, ¿qué debería hacer el algoritmo en caso de que se haya hecho una carga por la noche, pero por la mañana no se de tanta demanda prevista?, ¿Deberá descargarla a una hora no tan cara durante el resto de la mañana para volver a cargar en la siguiente orden del ciclo teórico? ¿O debería por el contrario no descargar esa energía barata descuidadamente?

Como se puede ver son preguntas no triviales, las cuales deberán tenerse en cuenta cuando se desarrolle esta futura implementación física, más el resto de cuestiones y complicaciones que salgan naturalmente durante el proceso. Se puede prever que la cantidad de trabajo que llevaría esta continuación del proyecto es análoga al ya expuesto en esta propia memoria. Este trabajo continuista requerirá su debida investigación formal y desarrollo, así como una memoria equivalente a esta propia.

Y eso son solo cuestiones del algoritmo, la parte lógica de este futuro proyecto. Además de la lógica también se deberá tener en cuenta como se hace el control de la batería eléctricamente. Si bien es el control de una batería clásica es un problema resuelto y con soluciones probadas, las particularidades de este

proyecto pueden requerir alguna solución personalizada y especifica. Solución que además deberá estar homologada y aceptada por el REBT (Agencia Estatal Boletín Oficial del Estado, 2025) y el resto de leyes que le apliquen.

Se dejan así estas cuestiones abiertas pero planteadas, listas para ser abordadas en un futuro por un proyecto similar a este.

CONCLUSIONES

La primera conclusión a la que se llega y la más inmediata es que **la hipótesis inicial era correcta**. Es posible **implementar una batería en casa y controlarla** de modo que pueda anticiparse a la demanda del usuario, **cargando en las horas en las que los precios sean favorables, y descargando cuando la demanda del usuario se dé en horas más caras**. El *software* desarrollado es capaz de anticiparse y calcular los ciclos correspondientes.

Sin embargo, también como se intuyó, el problema está limitado por el apartado económico. Las gráficas de guía generadas en el subapartado Resultados del Modo Histórico abordan este problema y lo solucionan, dando un rango de demandas en las que este software trabajará mejor. De los **datos disponibles** se deduce que este es aproximadamente el **límite interior de uso óptimo**. Es decir, **demandas mensuales medias superiores a 165 kWh**, o facturas superiores a **75 € mensuales**. Será a partir de estos valores donde este *software* tendrá más margen de maniobra y mejor funcionará.

El **mayor potencial** de este sistema ha sido al **combinarlo con una instalación de paneles solares tradicional**. No solo comparten tecnologías y partes, haciendo que el precio de la instalación se pueda justificar más fácilmente, sino que además existe una sinergia natural en el funcionamiento del conjunto. El principal problema de la energía solar es que no es “controlable”. Este *software* puede aportar una solución, puede hacer de la batería, un elemento normalmente pasivo, uno activo, más efectivo. El programa es capaz de adaptarse a diferentes condiciones de carga y distintas baterías elegidas por el usuario dinámicamente, generando un ahorro en el proceso.

Sin embargo, no se debe olvidar el problema que se está abordando, la predicción de demanda. Este es un problema muy complejo, y la solución que se le da con la IA si bien es funcional, mientras más avanzada sea esta IA, más exacto podrá ser el ciclo calculado. Por tanto, al ritmo que avanza esta

tecnología, es de esperar que en el futuro que estas mejoras repercutan y permitan mejorar aún más esta predicción, **mejorando la eficacia del *software*.**

El sistema de la **inteligencia artificial** fue costoso de implementar y complejo, y actualmente es necesario la lectura y documentación de cómo funcionan estas herramientas antes de poder implementarlas en el código y trabajar con ellas. Actualmente da un resultado aceptable, pero como ya se ha dicho, con la esperada continua mejora de esta reciente y novedosa herramienta, se considera que será posible mejorar este sistema en el futuro, tanto en eficacia como en facilidad de instalación y desarrollo.

En resumen, es posible instalar una batería y controlarla activamente para desfasar la demanda del usuario a las horas más baratas del día, pero su utilidad real está limitada no por el factor técnico, sino por el factor económico. Los usuarios con una demanda media o alta serán los que más beneficiados se vean de este *software* pues tendrán más potencial ahorro disponible para justificar la instalación de una batería. Además, es especialmente compatible con instalaciones solares, tanto por facilidad de instalación (solo se necesita comprar e instalar un controlador, algo bastante económico), así como por funcionamiento natural de los paneles solares, generando la energía más barata posible (gratis). Se ha de destacar también la abundancia solar en España, especialmente en el sur. Se concluye así que **el sistema puede ser viable** en la realidad.

BIBLIOGRAFÍA

Agencia Estatal Boletín Oficial del Estado. 2025. Reglamento electrotécnico para baja tensión e ITC. *BOE.* [Online] septiembre 03, 2025. https://www.boe.es/biblioteca_juridica/codigos/codigo.php?modo=2&id=326_Reglamento_electrotecnico_para_baja_tension_e_ITC.

Arduino. Arduino, Official Website. [Online] https://www.arduino.cc.

Autodesk. Autodesk AutoCAD: software de diseño y dibujo en el que confían millones de usuarios. [Online] https://www.autodesk.com/es/products/autocad/overview.

AutoSolar. 2025. 500W N-Type TOPCon Tensite. [Online] septiembre 10, 2025. https://autosolar.es/panel-solar-24-voltios/panel-solar-500w-n-type-topcon-tensite.

—. 2025. Batería Litio 4.8kWh Pylontech US5000 48V, especificaciones. [Online] septiembre 09, 2025. https://autosolar.es/baterias-litio-48v/bateria-litio-48kwh-pylontech-us5000-48v#specification.

—. 2025. Kit instalación placas solares Azzurro 6000 W con batería de litio de 5 kWh, ficha del producto. [Online] septiembre 09, 2025. https://autosolar.es/kit-solar-hibrido/kit-instalacion-placas-solares-autoconsumo-azzurro-6000w-con-bateria-de-litio-de-5kwh.

Battery University. 2023. BU-205: Types of lithium-ion. [Online] diciembre 08, 2023. https://batteryuniversity.com/article/bu-205-types-of-lithium-ion.

BatteryEVO. 2023. 24V EAGLE 60Ah 1.5 kWh. [Online] 2023. https://batteryevo.com/product/24v-eagle/.

Bereczki, T and Liber, A. 2024. The state of web scraping in the EU. *IAPP.* [Online] julio 03, 2024. https://iapp.org/news/a/the-state-of-web-scraping-in-the-eu.

Bishop, C. M. 2006. Pattern Recognition and Machine Learning. *Microsoft.* [Online] agosto 17, 2006. https://www.microsoft.com/en-us/research/wp-content/uploads/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf.

Box, G. E. P., et al. 2015. *Time Series Analysis: Forecasting and Control, 5th Edition.* 2015. ISBN: 978-1-118-67502-1.

Brown, T. B., et al. 2020. Language Models are Few-Shot Learners. *arXiv.* [Online] julio 22, 2020. https://arxiv.org/abs/2005.14165.

Consejo de la Unión Europea (Council of the European Union). Seasonal clock changes in the EU (Cambio horario estacional en la UE). [Online] https://www.consilium.europa.eu/en/policies/seasonal-time-changes/.

CVXPY. Convex optimization, for everyone. [Online] https://www.cvxpy.org.

eDistribución (Grupo Endesa). Consulta de consumo / Curvas de carga. [Online] https://www.edistribucion.com/es/servicios/gestion-suministro/consulta-consumo.html.

Entropy Survival / BatteryEVO reseller. 2025. BatteryEVO 24V Eagle Lithium Battery (60Ah - 1.5kWh). [Online] agosto 27, 2025. https://entropysurvival.com/products/batteryevo-24v-eagle-lithium-battery-60ah-1-5kwh.

ENTSO-E (European Network of Transmission System Operators for Electricity). 2017. Need for synthetic inertia (Technical note). *ENTSO-E.* [Online] 2017. https://eepublicdownloads.entsoe.eu/clean-documents/Network%20codes%20documents/NC%20RfG/IGD_Need_for_Synthetic_Inertia_final.pdf.

European Central Bank. 2025. Euro foreign exchange reference rates, USD. *ECB.* [Online] 09 septiembre, 2025. https://www.ecb.europa.eu/stats/policy_and_exchange_rates/euro_reference_exchange_rates/html/index.en.html.

García, Alberto Cruz. OMIEData (repositorio). *GitHub.* [Online] https://github.com/acruzgarcia/OMIEData.

GitHub. Build and ship software on a single, collaborative platform. [Online] https://github.com.

Goodwin, Michael. 2024. What is an API (application programming interface)? . *IBM.* [Online] abril 09, 2024. https://www.ibm.com/think/topics/api.

Google. Google Drive. *Almacena y comparte archivos en línea.* [Online] https://workspace.google.com/intl/es/products/drive/.

Huawei Technologies Co., Ltd. 2022. SUN2000 (2KTL–6KTL) Series User Manual. *Huawei.* [Online] marzo 2022. https://solar.huawei.com/-/media/Solar/attachment/pdf/au/service/residential/SUN2000-2-6KTL-L1-UserManual.pdf.

IEA PVPS. 2023 (informe, versión 2024/2025). National Survey Report of PV Power Applications in Spain. [Online] 2023 (informe, versión 2024/2025). https://iea-pvps.org/wp-content/uploads/2024/12/IEA-PVPS-2023-National-Survey-Report-Spain.pdf.

IEA-PVPS. 2024. Trends in Photovoltaic Applications 2024. [Online] 2024. https://iea-pvps.org/wp-content/uploads/2024/10/IEA-PVPS-Task-1-Trends-Report-2024.pdf.

International Energy Agency. 2025. Demand Electricity 2025 (analysis). *IEA.* [Online] 2025. https://www.iea.org/reports/electricity-2025/demand.

—. 2024. Spain Natural gas. *IEA (International Energy Agency).* [Online] 2024. https://www.iea.org/countries/spain/natural-gas.

—. 2024. World Energy Outlook 2024 Executive summary. *IEA.* [Online] 2024. https://www.iea.org/reports/world-energy-outlook-2024/executive-summary.

Jetbrains. PyCharm, The only Python IDE you need. [Online] https://www.jetbrains.com/pycharm/.

Ji, Z., et al. 2023. Survey of Hallucination in Natural Language Generation. *ACM Computing Surveys.* [Online] marzo 03, 2023. https://dl.acm.org/doi/10.1145/3571730.

Joselyn Stephane Menye, Mamadou-Baïlo Camara, Brayima Dakyo. 2025. Lithium Battery Degradation and Failure Mechanisms: A State-of-the-Art Review. *MDPI.* [Online] 2025. https://www.mdpi.com/1996-1073/18/2/342.

Kaplan, J., et al. 2020. Scaling Laws for Neural Language Models. *arXiv.* [Online] enero 23, 2020. https://arxiv.org/abs/2001.08361.

Kingma, D. P. and Ba, J. 2017. Adam: A Method for Stochastic Optimization. [Online] enero 30, 2017. https://arxiv.org/abs/1412.6980.

LeCun, Y., Bengio, Y. and Hinton, G. 2015. Deep learning. *Revista Nature.* [Online] mayo 27, 2015. https://www.nature.com/articles/nature14539.

Manembu, P. D. K., et al. 2023. A Systematicity Review on Residential Electricity Load-Shifting at the Appliance Level. *Revista: Energies (MDPI).* [Online] 2023. https://www.mdpi.com/1996-1073/16/23/7828.

Matplotlib. Matplotlib: Visualization with Python. [Online] https://matplotlib.org.

MEDREG (Mediterranean Energy Regulators). 2023. Energy price surge: Impacts and lessons learnt for Mediterranean energy markets (2023 update). *MEDREG.* [Online] 2023. https://www.medreg-regulators.org/Portals/_default/Skede/Allegati/Skeda4506-838-2024.4.4/ELE-WG-Price-Surge-Report.pdf.

Mohammed, A. H., Yousif, M. T. and al., et. 2020. Electricity load forecasting: a systematic review. *Journal of Electrical Systems and Information Technology, SpringerOpen.* [Online] septiembre 09, 2020. https://jesit.springeropen.com/articles/10.1186/s43067-020-00021-8.

Mosquera-López, S, M. Uribe, J and Joaqui-Barandica, O. 2024. Weather conditions, climate change, and the price of electricity. *ScienceDirect.* [Online] septiembre 2024. https://www.sciencedirect.com/science/article/pii/S0140988324004973.

National Renewable Energy Laboratory (NREL). 2025. Battery Lifespan. [Online] marzo 04, 2025. https://www.nrel.gov/transportation/battery-lifespan.

NVIDIA. CUDA Toolkit. *NVIDIA Developer.* [Online] https://developer.nvidia.com/cuda-toolkit.

OMIE (Operador del Mercado Ibérico de Energía). Precio del mercado diario. *OMIE.* [Online] https://www.omie.es/es/market-results/daily/daily-market/day-ahead-price.

Open-Meteo. Open-Meteo, Free Weather API. [Online] https://open-meteo.com.

pandas. pandas, pydata. [Online] https://pandas.pydata.org.

Pysolar. Pysolar. [Online] https://pysolar.org.

Python Software Foundation. Data model, Objects, values and types. *Python.org (Documentación oficial).* [Online] https://docs.python.org/3/reference/datamodel.html.

—. Python.org. [Online] https://www.python.org/.

—. Shelve, Python object persistence. *Python.org.* [Online] https://docs.python.org/3/library/shelve.html.

PyTorch. PyTorch Foundation. [Online] https://pytorch.org.

Raspberry Pi Foundation. Raspberry Pi, Official Website. [Online] https://www.raspberrypi.org.

Red Eléctrica de España (REE). 2025. Curvas de demanda y producción en tiempo real. [Online] 2025. https://www.ree.es/es/operacion/sistema-electrico/demanda-y-produccion-en-tiempo-real.

—. 2025. Red Eléctrica presents its report on the incident of 28 April and proposes recommendations. [Online] junio 18, 2025. https://www.ree.es/en/press-office/news/press-release/2025/06/red-electrica-presents-report-incident-28-april-and-proposes-recommendations.

—. 2025. Solar PV takes the lead in Spain's installed power capacity (Press release). [Online] febrero 4, 2025. https://www.ree.es/en/press-office/press-release/news/press-release/2025/02/solar-pv-takes-lead-spains-installed-power-capacity.

Srivastava, N., et al. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Revista: Journal of Machine Learning Research (JMLR).* [Online] 2014. https://jmlr.org/papers/v15/srivastava14a.html.

statsmodels. statistical models, hypothesis tests, and data exploration. [Online] https://www.statsmodels.org/stable/index.html.

Strubell, E., Ganesh, A. and McCallum, A. 2019. Energy and Policy Considerations for Deep Learning in NLP. *ACL Anthology.* [Online] julio 2019. https://aclanthology.org/P19-1355/.

Walker, A and Desai, J. 2022. Understanding Solar Photovoltaic System Performance. *U.S. Department of Energy (NREL).* [Online] febrero 17, 2022. https://www.energy.gov/femp/articles/understanding-solar-photovoltaic-system-performance-assessment-75-federal.

Xia, Y., et al. 2024. Understanding the Performance and Estimating the Cost of LLM Fine-Tuning. [Online] agosto 08, 2024. https://arxiv.org/abs/2408.04693.

Zhang, C., et al. 2017. Understanding deep learning requires rethinking generalization. [Online] febrero 26, 2017. https://arxiv.org/abs/1611.03530.

ANEXO, DOCUMENTACIÓN

Se añadirá a la memoria un capítulo de documentación complementaria a las propias *docstrings* ya existentes en el código. Tendrán una función similar, pero a un nivel de abstracción más alto y con un lenguaje menos técnico, con la intención de que sea más fácil seguir el flujo y la intención del código en cada paso.

Tanto las *docstrings* existentes en el código como esta documentación son autocontenidas, es decir, están redactadas para no necesitar una a la otra, para hacer que ambos el código y la memoria estén adecuadamente documentados. Sin embargo, debido a su naturaleza y enfoque a distintos niveles de abstracción y tecnicismo, se recomienda leer y consultar los dos documentos a la vez, es decir tener la memoria abierta y tomarla de referencia a la hora de leer el código. Aportarán distintos puntos de vista en el caso de que se encuentren dificultades al seguir uno u otro.

Esta documentación seguirá el flujo de la lógica del código. No se apegará perfectamente a la estructura en la que está escrito el código, pues por su propia naturaleza hace que dicho código no tenga un “orden” real, solo módulos con funciones dentro.

Como apunte final antes de empezar la documentación, y si bien se ha mencionado previamente, y será obvio para alguien con conocimientos técnicos solo viendo el código, se debe recalcar. El *software* está desarrollado y escrito íntegramente en *Python (Python 3.10* concretamente*)* (Python Software Foundation).

Main

El módulo *main.py*, como su nombre indica, es el script principal del código, y aquel que fusiona todas las funciones y subrutinas del código, las cuales se expandirán en sus respectivos apartados y subapartados a continuación.

main

Función *main* dentro del módulo *main.py*. Tiene 2 modos de funcionamiento, y se selecciona qué modo se usa leyendo un argumento que se le pasa al lanzar el código.

El **Modo Histórico** es el que toma datos de varios años del usuario y calcula para un rango discreto de precios de batería en el mercado, que tamaño de batería es la que minimiza el precio que pagaría el usuario. También calcula el ciclo de batería horario durante cada día de que tenga datos que justifica ese tamaño de la batería. Estos datos no son algo que necesite el usuario para tomar una decisión, pero el usuario puede elegir mostrar algunas soluciones concretas en las que se esté más interesado.

Los datos que, si necesita el usuario para tomar una decisión, y el resultado obtenido al ejecutar este modo es la gráfica que relaciona una capacidad de batería con un precio de mercado. Recomienda al usuario una capacidad de batería para un rango de precios de mercado de dicha batería. Mientras más cara sea la batería en el mercado más cara saldrá la inversión inicial y menos incentivo se tendrá para comprar una batería de mayor capacidad. Esta gráfica recomienda el tamaño óptimo de la batería, ni muy grande para que la inversión inicial no anule todo posible ahorro al optimizar el ciclo, ni muy pequeña para que el efecto de la batería sea despreciable sobre el global.

Cabe destacar que esto es la resolución de un problema puramente matemático, así que dará soluciones matemáticamente correctas, pero eso no asegura que de soluciones realistas. Así que no dará una única gráfica, pero 2. La primera será esa solución puramente matemática, yéndose a extremos tanto muy altos como muy bajos no factibles. La segunda gráfica es la misma que la primera, pero centrada en los valores centrales de dicha solución matemática, aquéllos cuya capacidad de batería calculada por el script en el modo histórico da valores en la realidad, ni excesivamente grandes, ni tendiendo a cero.

El **Modo Diario** hará un cálculo de optimización del ciclo de unos pocos días, se reutiliza el módulo de cálculo del modo histórico. Es en realidad un problema casi trivial una vez se ha resuelto el de históricos.

La complejidad de este modo no reside en el cálculo, sino en la obtención de datos. Es en esencia una predicción de demanda para el día de hoy. Y no solo demanda, también se tienen que predecir el resto de datos que se usan para el cálculo. Lo más destacable aquí es el uso de IA, tanto creación de los modelos como utilización es estos para predecir los datos. Es una parte compleja del

código, con su propio módulo dedicado a la IA, se profundizará en su respectivo apartado en este documento.

Este modo generará un fichero de texto plano (*.txt*) cada día en una carpeta, el cual contiene el ciclo horario de la batería predicho para optimizar el coste. También contiene algunos vectores adicionales como referencia que podría necesitar el sistema de control de la batería para "improvisar" y adaptarse durante el día con las directrices que le llegarían desde el fichero obtenido aquí.

Se ha elegido un archivo de *output* lo más simple y básico posible a propósito. De este modo se garantiza que la mayor cantidad de sistemas sean capaces de leerlo e interpretarlo, dejando disponible un gran abanico de opciones para hacer el control a tiempo real de la batería. Tales como usar un microcontrolador empotrado, microcontroladores sencillo y barato estilo *Arduino* (Arduino), o bien otras soluciones más complejas con más potencia, incluso soluciones con conexión a internet para monitoreo e interconexión como *Raspberry Pi* (Raspberry Pi Foundation).

En cuanto al código de la función *main* es realmente sencilla, simplemente empieza un conteo de tiempo, carga el archivo con toda la configuración (*Parametros.json*) y llama al modo que corresponda según el argumento. Una vez finalizado dicho modo detiene el conteo del tiempo y muestra por consola el tiempo transcurrido, confirmando la finalización del script correctamente.

Parametros.json

Antes de avanzar con el resto del código se debe hacer un inciso en el archivo de configuración. "*Parametros.json*" es un archivo de texto fácilmente editable en el cual se pueden introducir todos los datos y configuraciones necesarias para el script. Lo segundo que hace la función main luego de leer el argumento para el modo elegido es leer este archivo de configuración, y la mayoría de funciones, especialmente las de mayor abstracción beberán directamente de él.

Cuenta con los siguientes parámetros:

- ***param_batería_mercado***: son los datos de una batería del mercado, una lo más similar posible a la que se instalaría luego. Contiene a su vez:

  - *capacidad_bat_tipo*: la capacidad de dicha batería elegida. Se usará para tener un punto inicial de cuánto cuesta el kWh. Es un número (*int*), por defecto 1.
  - *precio_bat_tipo*: el precio en euros de dicha batería de referencia. Segundo dato para tener una referencia de coste del kWh. También un número (*int*), por defecto 200 (dando entonces un coste de 200 euros el kWh con los parámetros por defecto)
  - *potencia_carga_bat_tipo*: potencia máxima de carga de la batería en kW. Se asumirá que la batería elegida finalmente tendrá la misma potencia de carga, independientemente del tamaño final. Es un número (*int*), por defecto 1.
  - *potencia_descarga_bat_tipo*: potencia máxima de descarga de la batería en kW. Se asumirá que la batería elegida finalmente tendrá la misma potencia de descarga, independientemente del tamaño final. Es un número (*int*), por defecto 1.
  - *porcentaje_decimal_usable_capacidad*: porcentaje de la capacidad máxima de la batería que se usará en condiciones de trabajo normal. No se recomienda usar el 100% de la capacidad en un ciclo de trabajo nominal en baterías de litio (las más comunes) porque pueden acortar su vida útil (Joselyn Stephane Menye, 2025). Es un decimal entre 0 y 1 (*float*), siendo 0 no usar nada de la capacidad total, 0%, y 1 usar toda la capacidad disponible, 100%. Por defecto 0,65 (65%).

- ***param_usuario***: son los datos relativos al usuario y las condiciones de donde se instalará la batería.
  - *potencia_contratada*: potencia contratada en la instalación. Este dato pondrá un límite de cuánto se podrá consumir (e inyectar). Es un número (*int*), por defecto 4.
  - *target_years*: número de años que se espera explotar la batería, normalmente relacionado con la garantía que ofrezca el fabricante. En las baterías de litio suele rondar los 10 años en condiciones normales (National Renewable Energy Laboratory (NREL), 2025). Este parámetro definirá en cuántos años se puede dividir la amortización inicial de la compra de la batería. A más años más se podrá diluir dicha inversión inicial y más rentable será. Es un número (*int*), por defecto 10.
  - *multiplicador*: parámetro de *debug* y testeo. Multiplica la demanda por este parámetro, lo que hace que cambie la batería recomendada. A mayor multiplicador más consumo y más provecho se le sacará a la batería y se podrá instalar una mayor. Lo inverso ocurre a multiplicadores entre 0 y 1. Es un número (*float*), por defecto 1. No debería cambiarse para cálculos reales.

- ***rango_historicos_set_1***: Se puede dividir el set de datos de demandas en dos bloques. Útil para poder separar los datos, por ejemplo, un set para entrenar una IA, otro set para evaluarla.

  - *fecha_ini*: fecha de inicio de los datos del set. Es el set 1 así que la de inicio de todos los datos. Es una cadena de caracteres (*string*).
  - *fecha_fin*: fecha fin de los datos del set. Es el 1 así que donde lo quiera dejar. Desde eDistribución se pueden descargar dos años, por tanto, se recomienda hacer este set de un año. Es una cadena de caracteres (*string*).
  - *formato*: formato de las fechas que se han introducido previamente. Las 2 deben tener el mismo formato. Es una cadena de caracteres (*string*). Por defecto *"%d-%m-%y"*, ya que se han introducido fechas con la forma de *"01-03-23"*.

- ***rango_historicos_set_2***: El segundo set de datos, análogo al descrito previamente.
  - *fecha_ini*: fecha de inicio de los datos del set. Es el set 2 así que la continuación sin solape de la del final del set 1. Es una cadena de caracteres (*string*).
  - *fecha_fin*: fecha fin de los datos del set. Es el 2 así que el final de todos los datos. Es una cadena de caracteres (*string*).
  - *formato*: formato de las fechas que han introducido previamente. Las 2 deben tener el mismo formato. Es una cadena de caracteres (*string*). Por defecto *"%d-%m-%y"*, ya que se han introducido fechas con la forma de *"01-03-23"*

- ***param_solares***: datos de la instalación solar existente. En el caso de que no existiera dicha instalación con poner que hay 0 m^2 de paneles solares es suficiente.
  - *latitud*: auto explicativo, la coordenada de latitud de la instalación. Es un número (*float*). La latitud de Sevilla es 37,38283
  - *longitud*: auto explicativo, la coordenada de longitud de la instalación. Es un número (*float*). La longitud de Sevilla es -5,97317
  - *altura_metros*: auto explicativo, la altitud por encima del nivel del mar en metros de los paneles solares. Es un número (*float*). Por defecto 15,0.
  - *zona_horaria*: auto explicativo, la zona horaria de la instalación. La de Sevilla es *"Europe/Madrid".*
  - *eficiencia_porcentaje_decimal*: eficiencia decimal de los paneles solares (número entre 0 y 1, siendo 0 el 0% de eficiencia, y 1 un panel ideal, 100% eficiente). Un valor razonable de eficiencia para los paneles actuales suele ser 0,17 (17%), aunque aumenta constantemente según mejora la tecnología (IEA-PVPS, 2024). Es un número (*float*), por defecto 0,17.
  - *paneles_m2*: metros cuadrados de paneles instalados. Es difícil hacer una estimación aquí, dependerá mucho de cada usuario. En el caso de que no se tuvieran paneles solares bastaría con poner 0 m^2. Es un número (*int*). Por defecto 10 m^2, aproximadamente 4 paneles solares comunes.

    - *potencia_max_w_m2*: potencia máxima que puede generar el panel por metro cuadrado. Similar a la eficiencia, va aumentando constantemente según mejorá la tecnología, pero un valor razonable en el mercado actualmente es 250 W/m^2 (AutoSolar, 2025). Es un número (*int*).

- ***param_temperaturas***: datos de localización para el cálculo de temperaturas. Debería ser igual a los solares, pero se deja la opción libre.
    - *latitud*: referirse la latitud en los parámetros solares, idéntico.
    - *longitud*: referirse la longitud en los parámetros solares, idéntico.
    - *zona_horaria*: referirse la zona horaria en los parámetros solares, idéntico.
- ***opciones_calculo***: opciones más técnicas que modificarán algunos parámetros dentro del cálculo y del flujo del código y afectaran a cómo se muestran los resultados. Se recomienda una leve noción del funcionamiento del código antes de modificar estos parámetros.
    - *plot_intermedio*: principalmente de *debug*, aunque puede ser interesante para tener información adicional si el usuario sabe interpretarlo. Imprime al acabar el cálculo de históricos algunas soluciones concretas obtenidas. Es decir, el ciclo horario de todo el rango de datos para algunos valores de precio del kWh de la batería. Se pueden incluir varios, y buscará el más cercano en caso de que no exista algún valor concreto deseado. Es una cadena de caracteres (*string*). Por defecto *"100,200.5,300".*
    - *json_generales*: nombre y ruta del fichero donde se guardarán los resultados generales obtenidos del cálculo de históricos. En principio no debería haber motivo para cambiar esto a no ser que queramos cambiar la ruta. Es una cadena de caracteres (*string*). Por defecto "*resultados_generales_panel.json*"
    - *json_detalle*: Análogo al de generales. Por defecto "*resultados_detalle_panel.json*"
    - *paso_general*: incremento del precio de kWh de la batería en el cálculo general. Mientras menor sea este incremento (paso) más precisión se obtendrá, pero más tardará el cálculo. Recordatorio de que para mejorar la eficiencia se divide el cálculo en 2 fases, una general, con pasos más amplios, pero para ver la tendencia general, y otro de detalle, centrado en valores más interesantes. Este es el general, no debería hacerse demasiado pequeño si se quiere mantener la eficiencia, pero tampoco demasiado grande. Es un número (*int*). Por defecto 50.
    - *paso_detalle*: Análogo al general, pero esta vez para la fase del cálculo de detalle. Aquí si es interesante calcular más puntos a coste de más tiempo de cálculo, ya que será aquí donde realmente se elija la batería (aunque tampoco usar valores demasiado pequeños, no es necesaria tanta precisión). Es un número (*int*). Por defecto 1.

  - *rango_inicio_general*: basado en el precio del kWh de una batería tipo definido previamente, multiplica este valor por ese valor, obteniendo así un rango de inicio del modo histórico de cálculo en función de un precio tipo. Útil para definir el rango de inicio en función del precio tipo, por ejemplo, a mitad de este. Es interesante hacerlo por rangos en función del precio típico y no valores fijos porque cambiando ese valor tipo puedo variar dinámicamente el rango de cálculo automáticamente, y mantener la relación de cálculo constante. Es un número (*int*). Por defecto 0.
  - *rango_fin_general*: análogo al de inicio, pero siendo este el final del rango de cálculo. Es un número (*int*). Por defecto 3.
  - *capacidad_min*: Se ha comentado que se divide el proceso de cálculo de históricos en 2 fases una general, un problema matemático, y una de detalle, ya dentro del rango que tengan sentidos. Este es el parámetro que me marca el límite inferior del "tiene sentido". Baterías de capacidades inferiores a este valor en kWh serán consideradas inviables en la realidad y no entrarán en el cálculo de detalle. Es un número (*float*). Por defecto 0,5.
  - *capacidad_max*: Análogo al de capacidad mínimo, pero siendo este el límite superior, siendo consideradas estas baterías que superen este parámetro en kWh demasiado grandes como para ser fácilmente instalables en una vivienda. Es un número (*float*). Por defecto 5,0.

- ***rutas_modelos_IA***: rutas donde se guardan los modelos de las IA para el modo de cálculo diario. A no ser que queramos usar otros modelos, sobre todo en fases de *debug*, no habría motivo para cambiar estas rutas.
  - *modelo_demanda*: ruta del modelo de IA de predicción de demanda de la casa.
  - *modelo_solar*: ruta del modelo de IA de predicción de solar.
  - *modelo_precio*: ruta del modelo de IA de predicción de precios de electricidad de OMIE.

- ***batería_elegida***: parámetros de la batería elegida finalmente. Ya que no tiene sentido elegir una batería totalmente distinta de la calculada en el modo de históricos, las opciones en este apartado son limitadas, se asumirá que comparte la mayoría de parámetros con dicha batería de mercado tipo.
  - *capacidad_elegida_tot_kwh*: auto explicativo, capacidad de la batería instalada en kWh. Es un número (*float*). Por defecto 1,0 (aunque no tiene mucho sentido del dejar este valor por defecto, se recomienda activamente editar este valor al real).
  - *precio_final_batería_eur*: es el precio que costó la batería instalada (el parámetro inmediatamente anterior) en euros. Esto permite incluir la amortización diaria real en los cálculos de costes.

modo_historico

Uno de los 2 modos de funcionamiento del *software*. Tiene el objetivo de procesar una gran cantidad de datos para proporcionarle al usuario una gráfica con información de qué capacidad de batería es la más rentable para un rango de precios de mercado (un conjunto de parejas de valores, de capacidades en kWh y de precios en euros el kWh en el mercado). Como apunte, dadas las dimensiones de kWh y euros/kWh de las parejas de valores, es posible multiplicarlos para ver qué inversión en euros calcula el script que es óptima (no será un valor constante, también variará en función del precio del kWh).

El código está en una etapa de muy alta abstracción, aún se está en el rango de llamar subrutinas. Empieza creando la carpeta de trabajo, y le sigue la inicialización de los 4 valores clave que influirán en el problema de optimización:

- **Consumo del usuario**. Valores de demanda del usuario, para poder analizar el patrón de consumo y la magnitud del mismo. Es importante que sean los datos del usuario y no unos genéricos y aproximados ya que este código se basa en la personalización, una solución concreta para cada usuario. Para obtener estas curvas de demanda se recomienda descargarlos de eDistribución o algún servicio equivalente (eDistribución (Grupo Endesa)).
- **Precios**. Los precios de la electricidad en ese rango de consumos. Multiplicando la demanda (kW) por los precios (euros por kW) se puede obtener un coste en euros. En España, OMIE es la empresa encargada de fijar esos precios, y en su web hay históricos que se pueden descargar (OMIE (Operador del Mercado Ibérico de Energía)).
- **Potencias solares**. Al instalar una batería, una instalación solar suele venir de la mano, especialmente en el sur de España. Es interesante tener estos datos ya que será un escenario muy común. Además, con los paneles solares se puede potenciar el efecto deseado de esta optimización, podemos guardar el excedente de las horas punta de energía solar para las horas de precio máximo por la noche, controlando activamente la descarga de la batería para maximizar el resultado.
- **Temperaturas**. Si bien no es un parámetro que actúe directamente sobre los históricos, sí que será un parámetro clave para la predicción del modo diario, tanto los precios como las demandas están en función de la temperatura. No son necesarias en el cálculo de históricos, pero se inicializan por simetría con el modo diario.

Una vez inicializados (se tienen vectores de datos), lo siguiente es emparejarlos y unirlos en una tabla, creando así un *DataFrame* (una estructura de datos de *Python*, más sofisticado, y fácil de trabajar y procesar que una tabla “normal”, como la que podría usar Excel). Con este *DataFrame*

creado ya se tienen todos los datos necesarios, se empieza la optimización, la cual devolverá los valores emparejados que se esperan de este modo.

Antes de devolver estos datos y salir, la función tiene una entrada de una *flag* que da la opción de mostrar es pantalla la versión representada gráficamente de estos datos (*plot*). A no ser que se usen estos datos para otro objetivo, o sean entrada de otro código distinto, se va a querer siempre representar gráficamente los datos. Es la mejor forma de humanizar y entender fácilmente los datos obtenidos.

modo_diario

Uno de los 2 modos de funcionamiento del código. Tiene como objetivo predecir datos partiendo de unos históricos personalizados. Hará una predicción de unos pocos días en el futuro usando IA entrenada en los datos históricos de los 4 parámetros principales (referirse a *modo_historico*), y luego partiendo de una batería ya totalmente definida calculará el ciclo óptimo de demanda de la batería para el próximo día.

Por tanto, el inicio es igual que el *modo_historico*, obtiene los 4 datos principales y los empareja. El siguiente paso es obtener la “continuación” de esos datos. Para la temperatura es fácil acceder a webs meteorológicas que pueden hacer una estimación mucho mejor que la que se podrá hacer con una IA simple. El resto de parámetros se usará IA tomando como entrada datos previos del propio parámetro a predecir, más la temperatura, tanto la correspondiente a esos días previos, como la del día a calcular obtenida de dicha web meteorológica (Open Meteo (Open-Meteo) concretamente). Referirse a la sección de esta documentación correspondiente a la IA para más información.

Una vez obtenidos todos los datos futuros, análogamente a los históricos, se emparejan, y además se unen a los propios datos históricos, obteniendo así un *DataFrame* análogo al obtenido en el modo histórico, pero que llega hasta unos pocos días al futuro. A partir de aquí realizar una optimización de unos pocos días con una batería fijada es trivial comparado al problema previo. Se reutilizan los módulos de cálculo.

Una diferencia a apuntar es el resultado. El cálculo tiene dos salidas, una es el “cómo”, el ciclo, el otro es el resultado bruto y resumido. En este modo se quiere obtener el “cómo”, el ciclo que debe seguir la batería el próximo día para minimizar el consumo eléctrico del día. Y nuevamente a diferencia del modo

históricos, esta vez no es tan interesante representar gráficamente los datos, solo para *debug*. Solo se mostrarán por consola algunos datos generales. Aquí lo interesante es exportar el vector de ciclo para hacer que el controlador de la batería tenga directrices para el día. La parte de exportar en sí es simplemente un documento de texto plano (*.txt*), haciendo que sea fácilmente legible por la inmensa mayoría de dispositivos de control electrónico.

inicializar_consumos_historicos

Subrutina que leerá un fichero obtenido de **eDistribución** (eDistribución (Grupo Endesa)) (distribuidora de Endesa, tiene datos históricos de consumos horarios altamente precisos y en cantidad razonable, 2 años). Esta subrutina creará a partir de ese fichero *raw* una tabla procesada y fácilmente accesible, con una fila por fecha, y siendo cada columna una hora del día.

Además, ya que este proceso en condiciones ideales solo se deberá hacer una vez, guardarán esos datos ya procesados en una carpeta en formato *csv*. De este modo se hará un paso previo antes de procesar el archivo descargado de eDistribución, y antes buscará este *csv* que haya podido ser creado previamente, pudiendo saltarse el paso de procesado de datos, acelerando y optimizando el cálculo.

Este proceso de comprobar antes si existe un fichero *csv* ya procesado antes de volver a procesar el archivo de eDistribución ofrece una ventaja temporal mínima, ya que el procesado de este tipo de archivos es una tarea trivial para los ordenadores modernos. Pero este proceso tendrá más sentido en otras subrutinas que no puedan simplemente leer un archivo de entrada.

Al final esta subrutina simplemente apuntará al archivo *csv* creado o encontrado (guardará solo su ruta), no lo cargará en memoria aún. Eso solo se hará una vez se asegure la existencia de todos los datos necesarios posteriormente.

inicializar_precios_historicos

Estructura análoga a *inicializar_consumos_historicos*, pero cambiando el parámetro obtenido.

Es decir, primero buscará la existencia de un archivo ya procesado por una posible activación previa de la función. Si no existiera entonces procederá a obtener el parámetro objetivo de la fuente, luego crear una tabla ordenada y legible, y finalmente creará el archivo *csv* para su futura utilización. Tanto si ya había archivo *csv* o si acaba de ser creado, el output de esta subrutina será la ruta del archivo.

La diferencia es que ahora el parámetro es el precio, es decir, los precios horarios históricos de la electricidad. En España **OMIE** (OMIE (Operador del Mercado Ibérico de Energía)) es la encargada de esta función, así que la función hará un ***web scraping*** (sacar datos desde el propio código de la web, sin hacer una petición "formal") de la página de OMIE, y extraerá los precios horarios correspondientes a las fechas de las que hay datos de consumos históricos.

Se usan las fechas de los rangos de consumos históricos ya que será el único parámetro no "automatizable", es decir, se requerirá del usuario que realice el proceso de descarga manualmente, tal como ya se explica en su apartado. El resto de datos son búsquedas online, así que serán los consumos los datos limitantes.

inicializar_irradiancias_historicos

Nuevamente, estructura análoga a sus 2 subrutinas hermanas, referirse a *inicializar_consumos_historicos* y *inicializar_precios_historicos*. Busca *csv* del dato objetivo, si no lo encuentra lo crea leyendo la fuente. Al final retorna la ruta de dicho archivo.

La diferencia, similar a los precios, es cómo se obtienen los datos objetivos. En este caso se buscan las irradiancias, véase, la "intensidad" del sol, dato en W/m^2. Es "la cantidad de sol" que reciben los paneles por hora, y a más "sol" y a más paneles instalados, más energía se podrá obtener de la instalación solar. Para este dato se usará la librería de *Python* ***Pysolar*** (Pysolar), la cual entre otras cosas permite estimar la irradiancia horaria. Referirse a la sección del módulo de irradiancias para más información.

Igual al apartado anterior, se usará de referencia temporal los datos descargados de eDistribución.

inicializar_temperaturas_historicos

Como ya es la norma, esta función también comparte la misma estructura con sus otras 3 funciones hermanas comentadas previamente. Busca *csv* del dato objetivo, si no lo encuentra lo crea leyendo la fuente. Al final retorna la ruta de dicho archivo.

La distinción respecto a sus análogas es el dato buscado, ahora la temperatura. Es un dato fácilmente accesible, hay una gran cantidad y variedad de webs que ofrecen estos datos. Para el código se ha elegido la web de **Open Meteo** (Open-Meteo). Referirse a la sección del módulo de temperaturas para más información.

Como en los anteriores apartados, se tomará de referencia temporal el archivo de consumos del usuario.

buscar_archivo_regex

Durante las cuatro funciones de inicializar los datos se ha hablado siempre de un **proceso de buscar el *csv* primero**, y si existe ya se puede salir de la subrutina. Esta es la función que lo busca.

Toma como entradas el “dónde” es decir, en qué carpeta busca, y el “qué”, véase, qué debe buscar, qué patrón tendrá el archivo. También se puede optar por hacer que la función revise fechas, ver si son coherentes basadas en las existentes en el archivo de configuración, que las fechas del archivo encontrado estén dentro del rango de configuración. Es opcional se puede, o no, mirar fechas. Activando o desactivando esta funcionalidad permite alternar entre devolver los datos del archivo que tenga el nombre correcto, sin importar que contenga, a el mismo procedimiento, pero además comprobando que los datos que contiene son los esperados. Este primer “modo” debe ir seguido de una validación de datos posterior. Se usará este modo que no tiene validación implícita cuando se esté en la fase predicción de datos. En dicho modo no habrá unos datos esperados propiamente dichos, simplemente mientras más datos reales se obtengan mejor, pues así menos datos se tendrán que predecir y mejor será la precisión.

inicializar_vector_emparejados_historicos

Una vez se han ejecutado las cuatro funciones de inicializar y ya se ha confirmado que existen sus respectivos cuatro archivos de datos, localizados con sus rutas, se pueden **combinar todos en un único archivo**.

Aunque el primer paso es estandarizar fechas. Se lee el archivo de configuraciones y se elige el rango más amplio. Una vez se tiene esto, existe un módulo dedicado a esta tarea de emparejar datos, *emparejarEO*. Dentro de este módulo, se usará específicamente la función *emparejar_datos_historicos* para esta tarea. Referirse a su subapartado para más información.

Esta función devolverá un ***DataFrame*** (una tabla, pero optimizada para que la librería de *Python pandas* (pandas) trabaje con ella eficientemente) que contiene lo siguiente:

- **Cada fila es una hora**. Cada día son 24 horas, así que para 2 años son más de 8500 filas.
- **Columnas de las fechas y horas** previamente mencionadas.
- **Columna de contador del día**. Empieza en 1 para el primer día, y cada 24 horas (1 día) suma uno.
- **Columna del día de la semana**. 0 para lunes, 6 para domingo, con sus respectivos intermedios.
- **Columna del mes**. 1 para enero, 12 para diciembre, con sus respectivos intermedios.
- **Columna de hora**. Remanente del formato previo de 1 columna para cada hora, cada una nombrada H1 a H24. Se mantienen, pero repitiéndose para cada día.
- **Columna de *hora_int***. Similar al anterior, pero solo manteniendo el número, no la letra H. Representan la misma información, pero es más fácil trabajar con números sin la letra
- **Columna de precio**. Datos de precio de OMIE obtenidos con su respectiva subrutina.
- **Columna de demanda**. Datos de demanda del usuario de eDistribución obtenidos con su respectiva subrutina.
- **Columna de potencia solar**. Relacionado a los datos de irradiancia obtenidos con su respectiva función, pero ya procesados y calculada la potencia solar con los paneles solares instalados. Se realiza este cálculo durante el propio emparejado.
- **Columna de temperatura**. Datos de temperatura obtenidos con su respectiva subrutina.

inicializar_consumos_futuros

Esta es una función que solo se usa en el modo diario, pero por similitud de concepto, y por el flujo que sigue el modo diario, se documentará este set de funciones antes de seguir con el flujo del modo histórico.

En el modo diario, esta función parte de su análoga para los históricos, así que ya existe un documento de datos con archivos históricos. Pero esos datos acaban en una fecha concreta, para el modo diario se necesitan más datos. Idealmente hasta unos pocos días en el futuro. Pero las webs de históricos no serán capaces de dar datos a futuro, aparecerá la necesidad de usar herramientas de predicción de datos. El script está enfocado a predecir estos datos con IA, pero la IA no es perfecta, así que mientras es interesante apoyarse en esta funcionalidad, también se deben usar lo mínimo posible, solo los datos que no se tenga otra forma de obtener.

Y es en esta "filosofía" donde nace esta subrutina, **obtener datos reales a continuación de los datos históricos, la mayor cantidad posible**. Por el simple paso del tiempo, las bases de datos online donde se busca la informacion se irán actualizando y añadiendo datos reales, así que ahí se tiene una fuente de datos fiables adicionales.

Por tanto, la inicialización de datos futuros es el primer paso. Se deberá ver en qué fecha se quedaron los datos históricos, y seguidamente ir a la misma fuente donde se obtuvieron los datos históricos en su momento para obtener la máxima cantidad de datos nuevos posible.

Estos datos ya serán datos reales de la misma calidad que los históricos, así que se necesita una forma de **marcarlos como reales**, para primero no generarlos con la IA, segundo para no tener que buscarlos de nuevo en la fuente. Sería un proceso innecesario, se va a obtener el mismo resultado (son datos reales). Para días posteriores la búsqueda no empezará mirando en los históricos, en su lugar lo hará en estos últimos datos reales que se han obtenido ya, motivo de activaciones anteriores de la subrutina (ya que el modo diario se ejecuta todos los días, podrá obtener datos de un día adicional en cada activación de la subrutina). Se acentúa así la necesidad de marcar los datos para ver cuáles son datos reales y cuales necesitan ser generados o ya han sido generados "artificialmente". Se ha optado por la creación de un **archivo de fuentes**, hermano al de datos, con la misma estructura y mismas fechas, pero que, para cada dato del archivo de datos, el de fuentes lo identifica como real, marcado para generar con la IA, o bien ya generado con la IA.

Aparece así una distinción considerable de la subrutina de datos futuros respecto a la de históricos. Esta no creará un archivo y devolverá su ruta, en su lugar creará **dos archivos (datos, fuentes)** y devolverá sus dos respectivas rutas.

Sin embargo, hay un problema para el caso de los datos futuros de las demandas. No se pueden obtener automáticamente. De modo que, si bien la función está planteada y el esqueleto hace esa función, no es capaz de obtener datos nuevos reales, así que todo lo que hace es generar un archivo de datos a 0 (valor por defecto) y el de fuentes completo de marcas para generar con la IA.

inicializar_precios_futuros

Tendrá el mismo objetivo que su análoga de demandas. Primero buscará archivos creados, tanto su archivo de históricos, como los posibles archivos creados de una posible activación anterior. A partir de ahí buscará cuál es el último dato real que se ha obtenido de una base de datos de históricos, que puede ser simplemente el final del archivo de históricos, o la existencia de una pareja de archivos de datos y fuentes con algunos datos reales ya. A partir de ahí intentará obtener **más datos reales** de la base de históricos, en este caso la propia **OMIE**. Una vez que empiece a dar error o datos inválidos se considera que ya no se podrá obtener más datos reales hoy, así que se creará una pareja de ficheros de datos y fuentes (borrando los anteriores si existieran), marcando cuidadosamente qué datos son reales y cuáles hay que generar “artificialmente”.

La subrutina retornará las rutas de dicha pareja de ficheros, no se usará su contenido hasta que se tengan las parejas de archivos correspondientes a los 4 datos principales (referirse a *modo_historico* para más información sobre estos 4 datos).

inicializar_irradiancias_futuros

Análogo a las otras funciones de inicializar futuro, revisará qué datos existen ya, ya sea en forma de históricos o de archivos de datos futuros creados por activaciones previas de esta subrutina. A partir de ahí verá cuál es el último dato real que se tiene, y por tanto la base temporal para buscar **nuevos datos reales**. Análogamente a los datos de irradiancia históricos, la librería ***Pysolar*** será la

encargada de obtener estos datos (referirse a dicha función de *inicializar_irradiancias_historicos* para más información).

Una vez obtenidos estos datos se creará la pareja de ficheros de datos y fuentes actualizada y se borrará una posible pareja anterior, y se retornará la ruta de dicha nueva pareja de archivos.

inicializar_temperaturas_futuros

La estructura base sigue siendo igual a sus funciones hermanas, ver qué ficheros existen y qué datos reales contienen, ver cuál es el último dato real existente y a partir de ahí crear la pareja de ficheros de datos y retornar su ruta.

La diferencia clave en la temperatura respecto a las otras es que en la temperatura no se buscará en una base de datos de históricos. En su lugar Open Meteo, la web de donde se obtenían los históricos, ofrece una sección de ***forecast***, una predicción a futuro. Y es justamente esta predicción el objetivo final de esta subrutina. No solo se podrá obtener datos más precisos de este *forecast* de los que podría generar una IA sencilla como la de este código, sino que también se podrá usar esta temperatura ya predicha externamente como **entrada adicional de las IA** para los siguientes parámetros. Los datos de demanda y precio están fuertemente relacionados con la temperatura, son dependientes de ella. Tanto que modelos de predicción clásica usan la temperatura como entrada principal y solo unos pocos días previos para predecir el próximo día, tales como los modelos ARIMA (Mohammed, y otros, 2020).

Por tanto, los datos de temperatura predichos externamente, si bien siguen siendo una predicción, se pueden marcar como reales en el documento de fuentes. No existen herramientas en el código para mejorar esta predicción.

inicializar_vector_emparejados_futuros

La estructura de esta subrutina será extremadamente similar a la de *inicializar_vector_emparejados_historicos*, referirse a dicha sección para más información.

La principal diferencia es que para los datos futuros se trabajará con una pareja de ficheros, de fuentes y de datos. Por tanto, se crearán **dos *DataFrames* de valores emparejados**, uno para los ficheros de datos, otro para los de fuentes. Esta unión la gestiona su propia función dentro del módulo *emparejarEO*, *emparejar_datos_futuros* concretamente. Referirse a su subapartado para más información.

Los *DataFrames* generados tendrán exactamente la misma estructura que los de históricos, referirse al emparejado de históricos nuevamente para más detalle. A destacar que el fichero de fuentes no tiene valores numéricos, solo “letras”, indicadores de qué se debe hacer con dichos datos a los que apuntan estas fuentes.

combinar_historicos_y_presentes

Se han generado *DataFrames* idénticos para los datos históricos y para los datos futuros. Esto ha sido a propósito, esto facilitará el trabajo de esta subrutina.

Primero se debe verificar si efectivamente son iguales los *DataFrames*, por robustez. Segundo hacer que el contador de días mantenga la continuidad al unir. Es decir, sumar a este contador en los datos futuros el último día de históricos (continuidad). Tercero ya se puede **concatenar los *DataFrames***. El usar *DataFrames* de la librería *pandas* hace este paso trivial, existe una función que permite hacer esto. Por último, se hace una comprobación final de las fechas, que estén bien ordenadas, que no haya duplicadas ni faltantes, etc.

Nota importante, si bien esto deja un *DataFrame* construido, aún no se han predicho los datos faltantes. Referirse al módulo de IA para ver cómo se hace este paso, es el siguiente paso en el modo diario. Con la IA se completarán los datos faltantes, editando el *DataFrame* que se ha creado en esta subrutina, dejándolo preparado para el cálculo.

subrutina_mass_calc_optim

Subrutina de cálculo más compleja, solo usada en el modo histórico. El modo diario no necesita algo tan complejo.

Aquí no se calculará en sí aún, solo se **prepararán los datos para hacer recursivamente los cálculos** necesarios para cada punto de las gráficas finales objetivo del modo histórico. También lee configuraciones y prepara los archivos y carpetas auxiliares para el cálculo, tal como crear el entorno para usar una base de datos sencilla.

Primero lanza el cálculo general, que tomará un rango mucho más amplio, pero también usando pasos más amplios, pudiendo obtener así puntos de referencia. Con esos puntos de referencia ya se podrá comparar con los límites definidos en el archivo de configuración, pudiendo acotar así el rango de cálculo a algo más usable y con sentido. Se lanza así de nuevo la misma función de cálculo, pero ahora con un paso más pequeño, pero también en un rango acotado, obteniendo el cálculo de los detalles que se mencionaron en el modo histórico.

Para finalizar esta subrutina solo faltaría cerrar y borrar los archivos temporales de la base de datos, así como guardar los resultados en ficheros para futuras referencias. Y como los resultados se guardan en el disco, no necesita realmente dar un *return*. Lo único que retorna es el número de cálculos que se han realizado en total. No es de mucha utilidad, pero ya que se está monitorizando el tiempo, se puede obtener una media de cuánto tiempo se tarda por cada cálculo para mostrarlo por consola, por curiosidad.

subrutina_calculo_principal

Es la subrutina de cálculo llamada dos veces desde la subrutina anterior. Aquí ya se va enfocando el cálculo hacia la parte más técnica, sirviendo de conexión entre la subrutina y dicha parte técnica.

Se borran posibles archivos de resultados previos para no mezclar resultados, se seleccionan los datos a calcular únicamente de todos los que se tienen y se llama a la función de *problema_rango_precios* dentro del módulo de cálculo. Referirse al apartado de dicho módulo para más información.

obtener_rango_precios

En la subrutina *subrutina_mass_calc_optim* se habla de un paso entre los dos cálculos de acotar el rango de valores usables y con sentido. Es esta función la que lo hace.

En la configuración ya están predefinidos qué valores máximos y mínimos de capacidad de batería serán usables. El trabajo de esta función es tomar los valores de batería obtenidos con la función de cálculo general, eliminar lo que esté fuera de esos máximos y mínimos de capacidad y sacar así el rango de precios asociado a este rango que acaba de ser creado. O más bien el precio mínimo y máximo de este rango, es lo que usará la función de cálculo en detalle.

gestionar_ficheros_temporales

En todo el cálculo de históricos se ha mencionado el uso de archivos temporales y auxiliares. Esta es la función que los gestiona, tanto la creación al inicio, como el "resumir" dichos archivos en algo más fácilmente usable y borrar lo temporal al acabar.

Estos archivos temporales no son otra cosa que una **base de datos sencilla** estilo *shelve* (Python Software Foundation). Originalmente se usaban archivos *json* para el almacenaje de datos, pero la naturaleza iterativa de dicho cálculo hacía que se tuviera que abrir el archivo, y buscar y escribir en el fichero una gran cantidad de datos a gran velocidad. Esto es una tarea que simplemente ni el formato de archivo *json* (poco mejor que un archivo de texto plano estructurado) ni el disco duro están diseñados para hacer. Este proceso llegó a ser el cuello de botella del cálculo, el guardar los datos, lo cual es inaceptable y altamente ineficiente.

Identificado el cuello de botella el siguiente paso es solucionarlo. Y la solución es simple, es algo que ya lleva resuelto mucho tiempo en el mundo de la computación. Si se necesitan trabajar con grandes cantidades de datos rápidamente se usa una base de datos. Se opta por una base de datos estilo *shelve* como ya se comentó. Resumiendo, rápidamente, una base de datos *shelve* es un módulo de *Python* muy similar a una variable diccionario, pero con la propiedad de ser persistente (no volátil como una variable normal). Como ya se ve por este comentario rápido de cómo funciona destaca por su sencillez y similitud con la estructura *json* que ya estaba implementada (análoga a la de un diccionario). Y si bien es simple, funciona excepcionalmente bien en este

problema, consiguió bajar el tiempo de cálculo en un orden de magnitud entero respecto a los *json*. Tan efectiva es esta base de datos que hace viable el cálculo de optimización en procesadores poco potentes como el de portátiles relativamente antiguos.

En cuanto al código y todo el sistema de archivos temporales hay 2 fases, una de creación, al inicio del código, otra de borrado al final del código.

La de **creación** crea la carpeta, las propias bases de datos (una para los cálculos generales, otra para los en detalle) y un archivo de indexado. La función de este último será escribir en el que puntos han sido calculados en la fase de cálculo de detalles (solo la referencia a que el cálculo existe, no el cálculo en sí, el resultado del cálculo estará en la base de datos). La fase de detalle revisara en el archivo indexado si existe dicho calculo ya hecho. Si existe no necesita volver a calcularlo, solo debe referirse a la base de datos a recuperarlo.

La de **cerrar y borrar** los archivos temporales exportará las bases de datos a los archivos *json* iniciales (no tienen problema para almacenar la información en sí, solo escribir y borrar en ellos rápidamente, y son mucho más intuitivos de leer por parte del usuario) y borrará la carpeta entera de temporales, ya no es necesaria, y se borra todo de golpe así, no se necesita ir uno por uno.

subrutina_futuro_calc_optim

La subrutina de datos futuros **calculará una versión mucho más simplificada** de lo que ya se ha hecho en el modo de históricos, pudiendo saltarse así pasos e ir directamente a la función de cálculo. Aunque el hacer esto será necesario reducir el nivel de abstracción en esta subrutina.

El primer paso será la gestión de fechas, calcular fechas objetivo y máximos y mínimos y ver si las fechas a calcular están en el rango. Si es correcto se podrán filtrar los datos a solo los que se usarán en el cálculo (unos pocos días).

Segundo ya es el cálculo. Previamente se han comentado los problemas de la gestión de datos masivos, pero nuevamente este es un cálculo trivial, no se necesita almacenar nada, se pueden dejar los resultados almacenados en memoria en la variable. Se puede llamar así por tanto la función de *calculo_CPU* directamente del módulo de cálculo, y obtener así un diccionario de datos y

resultados calculados. Selecciona entre todos esos datos los datos del próximo día, y los retorna.

Estos serán los datos que se exporten al sistema de control en un fichero de texto plano tal y como se comentó en el modo diario.

Datos Endesa

Ya se han introducido en la sección del *Main* (en el modo histórico concretamente) los cuatro datos base que se usarán durante todo el código (consumo del usuario, precios, potencias solares, temperaturas). Durante el *main* también se han introducido los conceptos de inicializar dichos datos, pero en el *main* solo son subrutinas, las cuales llaman a sus respectivos módulos.

Este es uno de esos módulos, el de **consumos de energía del usuario, la demanda**. Este módulo se encargará de todo lo relacionado a dicho dato, desde obtener los propios datos, validarlos, ponerlos en el formato correcto, guardarlos y cargarlos de vuelta. También al final del módulo hay algunas líneas extras de código dedicadas únicamente al *debug* y a la creación del módulo (desactualizadas porque ya está creado e integrado en el *main*).

Como apunte, cuando se empezó a desarrollar el código se obtenían los datos de consumo desde la web de Endesa, viene de ahí el nombre del módulo. No fue hasta una etapa más avanzada del desarrollo que se cambió la fuente a eDistribución (eDistribución (Grupo Endesa)), mucho más fiable y con mayor volumen de datos, pero el nombre del módulo permaneció sin variar.

Se desarrollarán a continuación las funciones en este módulo:

crear_nuevo_archivo_edistribucion_historicos

La primera función que se llama desde el *main* al **inicializar los datos de consumo**. Como indica su nombre, su objetivo es crear un archivo que podrá ser usado fácilmente por otras funciones del código.

Esta función a su vez llamará a otras dentro de este mismo módulo, haciendo las veces de ***main* del módulo**. Para las demandas la entrada de datos se hace desde un documento descargado manualmente por el usuario desde la web de eDistribución. Por tanto, el primer paso será leer los datos de dicho archivo. Segundo, una vez leído se hará una validación y limpieza de los propios datos leídos. Y tercero y último ya se puede elegir un nombre y guardar ya estos datos validados en un archivo *csv* que poder leer a posteriori fácilmente.

cargar_datos_csv_edistribucion

Como se ha dicho previamente, el primer paso para obtener los datos de demanda es **leer el archivo de eDistribución**. Esta es la función que se encarga de esto. Es una función que como trabaja directamente con los datos no está a un nivel de abstracción tan alto como las subrutinas.

El archivo que viene de eDistribución tiene varias columnas: *CUPS*, *Fecha*, *Hora*, *AE_kWh*, *AS_kWh*, *AE_AUTOCONS_kWh*, *REAL/ESTIMADO*. Las columnas de *CUPS* y *REAL/ESTIMADO* son códigos e indicadores internos, no son necesarios para este cálculo. La columna *AS_kWh* (siglas de Activa Saliente, en kWh, refiriéndose a la potencia activa) es la energía que se inyecta en la red, la energía que "se vende" a la distribuidora. Es un parámetro importante, fuertemente relacionado con este programa, pero se opta por no usarlo ya que solo tendrá datos si ya se tienen instalados paneles solares, así que por redundancia y por escalabilidad no se usará. En su lugar se buscarán los datos solares por otras vías. Además, si la instalación solar no está integrada con el sistema de control de Endesa puede que este parámetro sea falseado. El parámetro que sí se usará, y el más importante de este archivo, es el de ***AE_kWh*** (siglas de Activa Entrante, en kWh, refiriéndose a la potencia activa). Esta es la demanda de la casa, la energía que la distribuidora proporciona al usuario, y por la cual se paga en la factura. Es el dato que se busca y además el más fiable, el más "tradicional". La columna de *AE_AUTOCONS_kWh* es simplemente la diferencia entre las dos columnas mencionadas previamente. Teniendo las otras dos, esta no aporta información nueva. Y por último las de Fecha y Hora, autoexplicativas. Lo único destacable es que la fecha viene en formato "*d/m/Y*" y que la hora es un número de 1 a 24 (o 23/25, según los cambios de hora)

Para el cálculo se seleccionarán por tanto las columnas de Fecha, Hora y *AE_kWh*. Son tres vectores de 24 valores por cada día (dos años). Lo primero que se hará es convertir la fecha. Sabiendo el formato de la fecha se cambiará a un tipo de variable más conveniente para *Python*. Lo siguiente es usar el propio campo de horas de 1 a 24 para que sean las columnas de la tabla, pasando así

de tres vectores de muchos valores, a una tabla de una fecha por fila, y 24 columnas (más los cambios de hora que introduce una hora 25, y la propia columna de fechas).

Sigue gestionar los cambios de hora, ya que eDistribución no los gestiona. Simplemente dice que un día tiene 25 horas o 23 y pasa al siguiente. El programa no tendrá la libertad de hacer eso, ya que mezclará datos de distintas fuentes, se necesita una base temporal robusta y compatible con el resto. Está el cambio de hora de “sumar” una hora en primavera, lo cual hará que se “salte” una hora, dando lugar a días de 23 horas. Y el cambio de hora de restar una hora en otoño, que hace lo contrario y por tanto aparecen los días de 25 horas. La solución del primer caso es una interpolación. No es lo ideal, pero son datos muy puntuales, es aceptable. El segundo caso es aún más sencillo, simplemente borrar las horas 25.

Queda así una tabla en el formato correcto y fácilmente usable posteriormente.

purgar_datos

Los datos obtenidos de eDistribución tienen lecturas faltantes en los últimos días. Se debe asegurar que la tabla hecha previamente tenga siempre los días con los datos completos, así que se hace una comprobación más y se empieza a revisar la tabla creada desde el final. Si la hora 24 (última hora) está a 0, se asume que este día está incompleto, se descarta entero aun si solo falta una hora, se borra la fila entera. Se repite el proceso en bucle hasta que la hora 24 no sea 0. Esto asegura que todos los días tienen **datos válidos y completos**.

decidir_nombre_edistri

Una vez ya se tiene la tabla creada y validada solo resta guardarla, pero para guardarlo hay que ponerle un nombre. Esta función se encarga de decidir el **nombre que debería tener el fichero**. Todos los ficheros de datos seguirán un proceso de nombrado estandarizado para poder identificarlos rápida e inequívocamente.

En realidad, es un proceso bastante simple, se decide de antemano un prefijo común a todos los archivos relacionados con este dato de demandas, *Datos_Edistribucion.csv*, y se mira en la tabla la fecha inicial y final. El nombre del archivo será la unión de estos tres datos (cuidando que el formato de la fecha sea el mismo siempre).

Se tiene así entonces tanto nombre como datos, se puede guardar el archivo.

crear_nuevo_archivo_edistribucion_futuros

Esta sería la función encargada de **completar los datos históricos** con más datos reales. Pero hay un problema, ya la obtención de los propios históricos fue un proceso manual, no se pueden obtener más datos de demanda automáticamente como requeriría esta función.

Queda entonces esta función aquí como esqueleto por simetría con el resto de módulos de datos y para una posible futura ampliación con esta funcionalidad si se obtiene una nueva fuente de datos de demandas del usuario.

Esta función ni está completa ni se usa. Referirse a las otras funciones de crear archivos futuros de los otros datos si se quiere ver una función de esta clase en funcionamiento, y como hubiera por tanto trabajado esta función.

Scrap OMIE

Otro de los módulos para gestionar uno de los 4 datos principales del script. Este como hace alusión su nombre es el encargado de la comunicación con OMIE (OMIE (Operador del Mercado Ibérico de Energía)). Es decir, es el **módulo del dato del precio de la electricidad**.

Como su versión análoga de Datos Endesa, el módulo se llamará desde el *main*, y trabajará con los datos directamente. Es el encargado de obtenerlos, validarlos, ponerlos en el formato correcto, guardarlos y cargarlos de vuelta. Y también como su módulo hermano, incluye al final unas líneas de código vestigiales

usadas para la creación y el *debug* del módulo, desactualizadas porque ya no se usan, están integradas en el *main*.

A destacar en este módulo es que OMIE no ofrece una forma fácil de acceder a sus datos, es decir, no tiene API (*Application Programming Interface*) (Goodwin, 2024). Muy resumidamente y simplificando, no se hacen peticiones a su web. En su lugar se piden los datos de un periodo concreto como si fuese un usuario normal, pero desde el código. La web entonces carga como lo haría normalmente, solo que el receptor en este caso no es un usuario humano normal, sino que en su lugar lo es este código, un ordenador. Y si un usuario es capaz de ver la información, entonces un ordenador podrá identificar la parte exacta en toda la información que proporciona la web que contiene los datos deseados y buscados. Se podrá aislar y descargar solo esta información concreta deseada. Se repite este proceso para todos los periodos que se requieran.

A esta técnica se le llama web *scraping*, y aún en esa breve explicación simplificada ya se puede intuir que es un proceso difícil, lento y tedioso, tanto de hacer como de ejecutar. Y además no es legal en todas las aplicaciones (particularmente en cuestiones de protección de datos y propiedad intelectual), aunque en este proyecto es perfectamente legal (Bereczki, y otros, 2024). Por suerte, existe en GitHub una librería pública llamada *OMIEData*, creada por Alberto Cruz García (García) que puede hacer el *scraping*, facilitando en gran medida el proceso, siendo así un pilar clave de este código.

crear_nuevo_archivo_omie_historicos

Nuevamente, simétrico con los otros datos, esta función es el ***main* del módulo**, aquella función que llama la subrutina en el *main*, y esta a su vez llamará al resto de funciones dentro del propio módulo.

El primer paso es simple, obtener los datos, no se puede hacer nada sin ello. Se hará un *scraping* tal como se comentó previamente, y tal como se entrará en más detalle en su función *omie_scrap*.

Una vez obtenidos los datos, y tal como pasaba con los datos de eDistribución. OMIE tampoco gestiona los cambios de hora, y aparecen días de 25 horas y días de 23. Así que se aplicará la misma solución, los días de 25 horas se borrará dicha hora 25, y los días con 23 horas se interpolará para encontrar la hora 24 faltante. No es la solución ideal, pero son solo unos datos muy puntuales, su efecto es despreciable en el cómputo total. Los datos obtenidos de OMIE son

más “limpios” que los de eDistribución, así que no es necesario hacer otras validaciones y purgas.

El siguiente paso, tal y como en eDistribución nuevamente, es guardar los datos ya obtenidos y limpios. Pero para ello se necesita un nombre. Se decide un nombre siguiendo el estándar ya creado y se guarda en archivo csv para su posterior uso.

La función retorna la ruta a la ubicación de dicho archivo creado, no los datos en sí.

omie_scrap

La función que **obtiene los datos**. Usando la función de *OMIEData* de GitHub se obtienen los datos en la fecha deseada (en el propio GitHub hay un ejemplo muy parecido al que se trata aquí).

La función hace un proceso iterativo y va obteniendo poco a poco y con varias llamadas a la web los datos. Es un proceso lento así que esto es una razón importante para hacer hincapié en la eficiencia a la hora de no obtener datos que ya se tengan.

Como apunte extra, si bien la función está bien mantenida y es perfectamente funcional, intuitiva y con ejemplos, al ejecutar la función da un par de *warnings* (avisos) durante la ejecución. Concretamente *FutureWarning*, mencionando el mensaje de error que el método usado dejará de ser compatible en un futuro, así que por precaución aplicó dentro de este módulo un par de “parches” a la función.

Los parches son una funcionalidad de *Python*, que hace que, al llamar a una función concreta en un módulo concreto de una librería, pueda sustituir su código por uno distinto durante la ejecución. Será básicamente el código original, pero sustituyendo las líneas donde se produce este aviso:

- ***patched_read_to_dataframe*:**
  El error de esta función está relacionado a la concatenación de valores vacíos. Actualmente el concatenar usado excluye los datos vacíos, pero en un futuro no lo hará. El propio mensaje de aviso da una posible solución si se quiere mantener la funcionalidad antigua, y es simplemente hacer en

un paso previo una exclusión de estos datos vacíos. Es una solución perfectamente válida, y será la aplicada. Primero se miran los datos válidos y se hace una lista de solo los válidos (no vacíos), y una vez que se tienen entonces concatena.

- ***patched_get_data_from_response*:**
  El error es similar al caso anterior, un aviso de que en el futuro el concatenar no excluirá automáticamente los datos vacíos y que si se quiere mantener la misma funcionalidad se deberán excluir estos datos en un paso extra antes de concatenar. Sin embargo, este caso es un poco más complejo y técnico. Se deben hacer unas comprobaciones y filtrados adicionales que se salen del objetivo de esta documentación. Referirse al código si se tiene interés en la solución concreta aplicada.

datos_omie_df

Función que lee un archivo *csv* y **crea un *DataFrame*** a partir del mismo.

Es en este punto donde la estructura ya es distinta de la de Endesa. La función de creación crea un archivo como su nombre implica, así que cuando se requiera usar esos datos se debe leer dicho archivo, esa parte es igual aún.

La parte diferencial es que es posible que en algún punto del proceso el archivo de datos puede haber sido borrado antes de usar dichos datos. Ya que los datos de precio se obtienen originalmente desde la web de OMIE, este escenario no es un problema, solo se necesita lanzar de vuelta la función de *omie_scrap*, con todo el conjunto de *scrap*, decidir nombre y guardar. Solo que ahora no se retorna la ruta, ahora son los datos lo que se buscan, el *DataFrame*.

buscar_datos_scrapeados

Función del modo diario. En este modo se ejecuta un ***scrap*** de datos desde los últimos datos que se tengan **hasta el más reciente** que se pueda obtener, es decir, hay que ver que rango de fechas son reales y cuáles hay que generar, y usar este segundo rango para *scrapear* OMIE.

En el modo futuro se trabajan con parejas de archivos (uno de datos, otro de fuentes), así que leyendo e interpretando este archivo de fuentes se pueden obtener los rangos de fechas deseados.

El primer paso será leer el archivo, seguido de una validación de datos y conversión a un formato más conveniente. Siguiente, ya que trabajar con horas individuales no tiene sentido en esta tarea, se resumirán las 24 horas del día a un único valor representativo. Si todas las horas del día están marcadas como reales, se considera el día entero como real. En caso de que al menos una no sea real, se toma como que se debe generar de vuelta el día entero.

De modo que ya se tiene un vector, ahora solo resta tomar la fecha mínima y máxima de cada rango de datos reales y a generar respectivamente. Usando estas fechas se puede generar ahora una máscara sobre el archivo de los datos. La máscara se usará de localizador y ver qué datos se deben recuperar, creando así un *DataFrame* de datos de precios eléctricos incompleto, pero ya con algunos datos que no será necesario volver a obtener, yendo en sintonía con lo comentado en otras ocasiones de eficiencia y de no volver a obtener datos que ya se tienen.

Además, ya que este paso le seguirá la creación de una nueva pareja de archivos de datos y fuentes más actualizadas, se pueden borrar la pareja de datos anterior para mantener el orden.

Se retorna entonces las fechas iniciales y finales de los datos a *scrapear* nuevos, tanto como los datos que ya se tienen y que serán combinados a lo *scrapeado*.

crear_nuevo_archivo_omie_futuros

También una función del modo diario, y la inmediatamente posterior a la previamente documentada de buscar los datos ya existentes.

Su nombre ya indica qué hará, la creación de la **pareja de archivos** de datos y fuentes que se usan para **gestionar los datos futuros**, uno de datos y otro de fuentes. El archivo de datos es el más intuitivo, se buscan datos y se guardan los datos, sin más. Muy similar al proceso de la creación del archivo de los históricos. Es en el de las fuentes en el que radica la novedad. Tiene exactamente el mismo formato que su archivo gemelo de datos, pero no contendrá valores numéricos en su interior. En su lugar tendrá *flags* y palabras

que indicarán la fuente o qué se debe hacer con ese dato asociado. Cada dato tiene una fuente asociada.

La base inicial para la creación serán los propios históricos y la última fecha de los históricos, y su objetivo último e ideal es obtener todos los datos reales desde dicha fecha hasta unos pocos días a futuro desde la activación del código. Una tarea obviamente imposible de base, pero la verdadera intención es acercarse lo más posible a ese ideal, obtener la máxima cantidad de datos reales posibles.

En el caso de los precios, la fuente de datos es OMIE, así que se hará un *scrap* desde esa fecha última de históricos hasta la máxima posible, hasta que de error. Se le hará un procesado de datos análogo al que se hacía a los históricos (gestionar cambios de hora tanto de horas extras como faltantes, borrando o interpolando respectivamente, y luego se quitan filas con huecos inválidos hacia el final.

Una vez se tienen estos datos se le asociará su fuente, marcándolos como datos reales. Se usa el mismo formato. Se copia la columna de fechas y a estas fechas se le insertas las columnas de horas, obteniendo así un archivo vacío listo para rellenar. En principio se rellena entero como si todos los datos fueran reales, luego se sustituyen por su etiqueta correcta.

Sigue la parte de los datos faltantes. Previamente se han purgado los datos incompletos, por lo que el *DataFrame* de datos solo llega a una fecha. Pero es conocida la fecha objetivo que debería haber llegado idealmente. Se rellena cada hora en esta diferencia entre última fecha con datos reales y fecha última objetivo con “0” en los datos (podría ser cualquier valor en realidad). Se hace un proceso similar en las fuentes, pero aquí radica la clave, y es que ahora las fuentes se rellenan con “*AGenerar*”. Se identifican los datos que faltan con el archivo de fuentes y así ya serán fácilmente identificables cuando se generen externamente, quedando claramente marcados del resto los datos incompletos.

Y volviendo a la estructura clásica, restaría decidir los nombres de los archivos y guardarlos, para retornar sus rutas (son dos archivos, 2 rutas).

Sin embargo, durante el código se ha obviado un paso más por mantener el flujo más sencillo y legible. Dicho paso sería el dónde entra la función anterior de buscar los datos ya obtenidos previamente en otras ejecuciones del código. Esta función toma como parámetro también el *DataFrame* generado por dicha función anterior, y justo después de *scrapear* y filtrar y validarlos no se generan los datos de fuente. En su lugar se **suman estos dos *DataFrames***, el nuevo y el que ya tenía (teniendo cuidado con el orden, y que la fusión de datos quede coherente). Una vez que se tienen los datos reales completos es entonces donde se crean

las fuentes y se marcan a estos datos como “*Real*”, estos si son los datos al completo.

Nótese que los datos de consumo que no se han podido obtener se rellenan con 0 y se marcan para generar en el futuro, pero no se ha hablado en ningún momento en el cómo se generan. Y eso es porque aún no es el momento de completar los datos. Para completar los datos faltantes se usará IA, referirse a su módulo para más información, pero el uso de la IA es una de las últimas etapas del cálculo, tal y como ya se dijo en el *modo_diario* del módulo *Main*.

Como curiosidad, si se lanza este código en las últimas horas del día, OMIE devolverá los **datos reales del próximo día**. Esto no es debido a un error, o que OMIE de una predicción de mercado. Son los datos reales de precio que habrá ese día (menos algún cambio urgente de última hora).

Para entender este funcionamiento se debe entender cómo funciona el mercado eléctrico español. Simplificando, OMIE es la empresa encargada de casar la demanda predicha con las ofertas que las generadoras ofrecen. Es decir, encuentra la forma óptima de cubrir dicha demanda al menor coste posible con las ofertas que tiene a su disposición. Esto determina el coste de la electricidad.

Este proceso se da el día anterior al objetivo a media mañana (hora peninsular), así que por la tarde la propia OMIE ya ha actualizado la base de datos en su web con los precios del siguiente día. Aún no ha ocurrido, pero ya se han fijado, y son totalmente fiables porque es la propia OMIE quien los ha fijado.

Datos Solares

Script para el tercer dato principal, los **datos solares (de irradiancia).** Nuevamente, la estructura es muy similar a las anteriores, especialmente a los consumos, debido a que igual que dicho dato, también se obtienen automáticamente.

Y también análogo a los anteriores se incluyen al final unas líneas vestigiales de cuando se creó el módulo. Están desactualizadas ya que el módulo está integrado en el *main* ya, pero quedan a modo de *debug*.

Este módulo está basado en la librería *Pysolar* (Pysolar). Es una colección de otras librerías, todas de *Python* y en código abierto. Al ser una colección de librerías no hay un único autor, pero como referencia base se puede nombrar a la persona que las unió, el usuario *pingswept* en GitHub.

Los datos solares que se obtendrán y guardarán serán irradiancias, es el dato que se puede obtener. Sin embargo, en el problema no habrá cabida para este dato, no es útil. Se quiere en su lugar la potencia generada por los paneles solares. Se incluye una función de "traducción" de un dato a otro, pero solo se aplicará justo antes del cálculo, en este módulo se trabaja casi íntegramente con irradiancias solares, la "cantidad de sol" que hace, o en palabras más técnicas, la potencia solar por metro cuadrado que llega a la superficie desde el sol.

crear_nuevo_archivo_solar_historicos

Análogamente a las funciones hermanas ya documentadas, esta es la función que se llama desde el *main*, y hace las veces de ***main del módulo***, al menos del modo histórico.

Llamará a otras funciones para obtener los datos, los limpiará y ordenará, luego los convertirá en una tabla para seguir el formato estándar, y por último guardará los datos para futuras referencias.

Devolverá la ruta de dicho archivo guardado.

obtener_irradiancia

Primer paso real del proceso, no se puede avanzar sin los datos. La librería *Pysolar* es capaz de estimar la **irradiancia solar en unas coordenadas** a una altura concreta a la hora especificada. Además, ya que las horas son una entrada se necesita generar un vector de fechas (y horas) adicional entre las fechas especificadas como mínima y máxima.

Con otras librerías de *Python* también se puede localizar la hora para acomodar los cambios de horas para facilitar el emparejado de datos de irradiancias y

futuras potencias solares con los datos ya obtenidos de demandas y precios de electricidad.

Cabe destacar que estos datos son una estimación. El cálculo solar es un problema muy complejo, se necesita una gran cantidad de datos y técnicas estadísticas y matemáticas avanzadas, hay una gran cantidad de literatura abordando este problema. Pero en el caso del problema en total en conjunto que aborda este script, esta aproximación es más que suficiente, no es necesario perder tiempo y recursos de cálculo obteniendo unos datos solares más precisos. Esta es una de las varias concesiones que se hace durante el código en pos de eficiencia y usabilidad por equipos menos potentes.

Al final devolverá las fechas y las irradiancias, dos vectores emparejados.

crear_df_tabla_irradancias

Tomando las salidas de la función inmediatamente anterior, la de obtener_irradancias, se **convierten los vectores** de fechas (y horas) y sus datos asociados a un ***DataFrame***, una tabla, con el mismo formato que los análogos de los datos de demandas y precios de electricidad.

El procedimiento es similar, crear las columnas de H1 a H24, y agrupar los datos del mismo día en dichas columnas, pasando de formato vector a formato tabla. Más compacto a la hora de guardarlo y más fácil de leer por el usuario si lo quisiera (mejora así también la depuración del código al “humanizar” los datos).

Parecido a los datos de Endesa, se añade una función de purga de datos al final de la creación de esta tabla, pero también una de falseo de datos, que se activarán complementariamente. Es decir, si se usa una no se usará la otra, pero siempre se usará una de las dos. Más información de lo que hacen a continuación en sus respectivos apartados, pero, en resumen:

- El **falseo de datos** es para el modo histórico, en el caso de que se fallara en obtener los datos más recientes se hará otra concesión y se “inventarán” algunos días faltantes para poder completar el cálculo a costa de más imprecisiones (solo serán unos pocos días, despreciable en el cómputo total).
- La **purga de datos** es para el modo diario. Si se falla en obtener los datos de algún día como antes, ahora se pueden borrar. Se va a usar una herramienta mucho más precisa para generar datos faltantes (IA) dentro del flujo del código.

falseo_datos

Toma como entrada un *DataFrame* con posibles días inválidos al final. En condiciones ideales se deberían borrar, pero si se borran se pierden días de cálculo, y los datos de demanda son demasiado valiosos para “desperdiciarlos”. En su lugar se **falsean los datos inválidos**. Se toma el último día con datos correctos y se copia dicha información en los días que no fueran válidos.

Es una concesión razonable que se realiza, ya que en rangos de tiempo cortos (inferior a una semana) el ciclo solar es muy similar. Retorna el *DataFrame* con los datos “completos” (falseados).

purga_datos

Este es mucho más simple. Se ejecutará en el modo diario, el cual usará IA para predecir datos faltantes. Entonces, tomando un *DataFrame* de datos de irradiancia de entrada con posibles datos incorrectos, se puede delegar a la IA el trabajo de generar los datos faltantes, aquí simplemente se pueden **borrar los datos inválidos**, ya se generarán mejor posteriormente.

guardar_irradancias_csv

Para finalizar, simplemente toma el *DataFrame* con los datos de irradiancias solares ya en el formato correcto y los **guarda**.

Pero antes decide un nombre para el archivo alineado con los guardados ya previamente. Retorna la ruta del archivo creado.

datos_solar_df

Tal y como ya se hizo en los datos de OMIE, los **datos solares se pueden obtener automáticamente**, así que se añade una función que no solo sea capaz de leer los datos guardados creados por las funciones de este módulo, y que también pueda identificar si no existe el archivo, pudiendo así lanzar las funciones necesarias para volver a crearlo, lanzando la función *crear_nuevo_archivo_solar_historicos*.

Tanto si se tenía el archivo como si se ha tenido que crear uno nuevo, el objetivo último de esta función es leer dicho archivo y sacar del mismo un *DataFrame* con los datos de irradiancias solares en formato de tabla, más fácilmente usable por *Python*, y el cual será la salida de esta función.

buscar_datos_scrapeados

Lleva exactamente el mismo nombre que la función de OMIE, y como se ha de suponer hace exactamente lo mismo. Solo que esta por estar en el módulo de solar trabaja con datos de irradiancias. Se dará una versión más resumida de la función, referirse a la función del mismo nombre en el módulo de OMIE para más información.

Es una función exclusiva del modo diario cuyo objetivo es mejorar la eficiencia general del código haciendo primero una **comprobación de que datos se tienen** ya para no volverlos a generar u obtener nuevamente. Mejora así por tanto la velocidad y uso de recursos.

En el modo diario se trabaja con parejas de archivos, uno de datos y uno de fuentes. Entonces si se quiere saber qué datos se tienen solo bastaría con ver que datos están marcados como reales en las fuentes. Luego se obtienen los datos relacionados a esas fuentes, y el rango de fechas en los que se tienen datos y en los que se deben intentar obtener más datos aún. Se genera así por tanto un *DataFrame* con datos solares incompleto, y un rango de fechas objetivo que se debe completar.

Y no se debe olvidar borrar los ficheros antiguos, se van a generar unos nuevos más actualizados de ambos datos y fuentes, los previos quedaran obsoletos.

crear_nuevo_archivo_solar_futuros

Nuevamente y manteniendo una simetría entre rutinas, el paso siguiente a obtener los datos solares vuelve a ser el generar los archivos de futuro, la pareja de datos y fuentes. Su estructura será análoga a la de *crear_nuevo_archivo_omie_futuro* del módulo de OMIE, simplemente se usará la función de obtener datos "local", es decir, la función de *obtener_irradiancias*. Referirse a dicha función si se desea entender la lógica de esta función en detalle.

El **archivo de datos** como su nombre hace alusión es el cual contiene los datos qué se han obtenido o se obtendrán. El de **fuentes** es una marca para saber de dónde vienen o que se deberá hacer con los datos. Cada dato tiene su correspondiente fuente asociada en un archivo gemelo.

El objetivo final será partir de una fecha y unos datos iniciales, y obtener la mayor cantidad posible de datos hasta unos pocos días en el futuro. Pero por el planteamiento imposible del problema, el verdadero objetivo será acercarse lo máximo posible y que una IA u otro método de predicción termine el trabajo.

No se parte de cero, se parte de al menos unos datos históricos de irradiancias, y muy posiblemente la función anterior de buscar datos también haya encontrado datos previos posteriores a los históricos que se podrán reusar, no hará falta buscarlos dos veces. Con eso ya se parte de una base de datos sólida, un *DataFrame* inicializado e incompleto, el cual se va a completar.

Los datos solares se estimarán usando la librería *Pysolar*. Se podría esperar que ya que es una estimación podría aproximar los datos futuros, y si bien se acerca mucho al presente, incluso esta librería tiene sus límites. Como apunte recordar que esta función tenía dos comportamientos respecto a los datos inválidos, uno que era falsearlos, usado en el modo de históricos, y el otro, el cual es el que se usa aquí en el modo diario, es el de purgar datos. Simplemente borrará los datos que no sean válidos, ya que como se ha mencionado previamente, existen métodos en el flujo del script en el modo de diario para generar este tipo de datos.

Una vez se obtiene la cantidad máxima de información real posible de datos de irradiancia, sigue la decisión de qué hacer con los que no se tienen. La respuesta es simple. Rellenar con 0, marcarlos como a generar posteriormente y dejar que la IA o el método predictivo elegido finalmente lidie con dicho problema. Pero eso será en el futuro.

Por el momento es suficiente con este paso previo de rellenar con 0 y marcar. Se tendrán unos *DataFrames* lo más completos posibles en sus respectivos archivos de datos, se procederá a decidir unos nombres basados en el estándar ya usado varias veces durante el resto del código y se guardará la pareja de archivos en formato *csv* en una tabla con un día por fila y horas diarias por columna. Se retornarán las rutas de dichos archivos.

calculo_paneles

Durante todo este módulo se han hablado de irradiancias. Pero también se ha dicho que las irradiancias son inútiles en el cálculo en última instancia, se necesita la potencia generada por los paneles solares cuando se vaya a hacer el cálculo. Esta es la función que une esas 2 afirmaciones, la encargada de **calcular la potencia que generan los paneles a partir de la irradiancia**.

Si durante la obtención de la irradiancia ya se hicieron concesiones en la obtención de los datos debido a la complejidad del problema en sus versiones más exactas, en este punto el problema no se hace más sencillo. Se hace considerablemente más complejo, aparecen aún más variables y modos de interactuar de la luz solar con el medio. Simplemente no es factible hacer tal cálculo a ese nivel de detalle en este script, cuya finalidad no es obtener un cálculo solar preciso.

En su lugar se vuelven a hacer más concesiones y aproximaciones. Se reduce tal problema a varias multiplicaciones. Primero, se parte de irradiancias, es decir, W/m^2. De la propia unidad ya se ve que se necesitarán metros cuadrados. Estos se obtienen de la instalación del cliente. Se lee el archivo de configuraciones, es uno de los datos que aparecen ahí. Con esto, se puede pasar de W/m^2 a W que inciden en los paneles solares del usuario (sin tener en cuenta parámetros como reflexión indirecta, sombras, ángulos etc. Nuevamente, esto es un cálculo aproximado). Pero los paneles no son capaces de absorber toda la energía solar que incide sobre ellos, eso implicaría una eficiencia del 100%, cosa que la tecnología solar aún está muy lejos de estar remotamente cerca. De hecho, la eficiencia de un panel solar comercial estándar ronda entre el 14% y el 22% (IEA-PVPS, 2024). Se puede aplicar así esa eficiencia, definida mediante un parámetro de configuración, a la energía total “entrante” para calcular la energía total “saliente”, también en W. Este dato ya es una potencia solar útil, es el dato que se necesita para el cálculo. Solo que aún no, no se puede devolver este dato. Se han calculado W, pero el estándar del *DataFrame* son los kW. Se hace una conversión final de W a kW y se devuelve el vector de potencias solares obtenido a partir de la configuración del usuario y la irradiancia solar.

Datos de Temperatura

Módulo encargado de **gestionar** el cuarto y último dato principal, las **temperaturas**. Aunque es un dato que solo tendrá utilidad en el modo diario. Se gestiona como el resto de datos, y la estructura será análoga al resto de módulos de gestión de estos datos, pero no se usarán los datos obtenidos de temperatura en el modo histórico, aun cuando se tienen.

Esto es debido a que **la temperatura no influye en el cálculo en sí, pero afectará directamente a los otros tres parámetros**. Es decir, demanda, precios y energía solar son funciones de la temperatura (aunque la solar en menor medida, su relación es mucho más débil).

Se puede intuir por tanto que la temperatura será un parámetro clave cuando se tengan que predecir los otros parámetros, así que se realizará un esfuerzo extra para obtener unos datos de temperatura lo más fiables posible. Los datos futuros de temperatura no se generarán con una IA o alguna otra técnica, se hará una petición a una web meteorológica en su lugar (Open Meteo (Open-Meteo), la cual tiene un apartado para archivos y otra para predicción, ambas gratuitas y de fácil acceso con su API), que aportará unos datos de mucha mejor calidad de lo que se podrá calcular jamás con los datos que se usan en este código. Además, la temperatura es la entrada para estimar los otros parámetros, aun si se quisiera estimar con la IA no se podría fiablemente, no hay entrada para la temperatura aparte de los históricos.

Pero aun con estas variaciones, la estructura del código sigue siendo en esencia la misma que la de sus parámetros hermanos, solo cambian algunas funciones internamente. Y manteniendo la estructura, también se mantienen al final del código unas ideas vestigiales de cuando se creó el módulo, ya desactualizadas.

crear_nuevo_archivo_temperaturas_historicos

Función muy similar a sus análogas de los datos anteriores, hace las veces de ***main* del módulo** para los datos históricos.

Su estructura también es simple, simplemente llama a la función para obtener los datos (históricos) llamando a *obtener_temperaturas*, luego se pasa al formato que ya se ha estandarizado para dichos datos con *crear_df_tabla_temperaturas*, y por último se guardan dichos datos usando *guardar_temperaturas_csv*.

Retorna la ruta de dicho archivo recién creado.

obtener_temperaturas

La función para **obtener los datos de temperatura** que se llama durante todo el módulo cada vez que se necesitan datos **históricos**.

Esta función se basa en la API de Open Meteo, la cual devuelve de sus registros históricos los datos deseados en las coordenadas deseadas.

Como paréntesis, y simplificando mucho, una API (*Application Programming Interface*) es el conjunto de "reglas" y estándares que usan las webs y otros programas informáticos para comunicarse entre sí. Por ejemplo, si una aplicación necesita cierta información de una web, puede simplemente preguntarle por dicha información y la web se la proporcionará fácilmente, sin necesidad de hacer tantos pasos y tan tediosos como en la técnica de *scraping* (el método usado en OMIE). Además, esto también supone menos recursos a la propia web que facilita estos datos. La web de Open Meteo dispone de una API gratuita de uso público, así que mientras se formule correctamente, se podrá generar una petición a Open Meteo, la cual la gestionará y responderá. Se tiene así un acceso fácil a la extensa base de datos que tiene dicha web con solo unas pocas líneas y tomando muy poco tiempo ya que la "respuesta" contendrá un único bloque con los datos requeridos. No hay necesidad de iterar para obtener los datos (nuevamente, como se hacía con la técnica de *scraping* en OMIE).

La petición en si es sencilla, simplemente se les da el formato correcto a las fechas inicial y final, para luego crear un vector de horas localizado (ya incluyendo los cambios de hora de España). Luego se introduce la dirección de la API y se acompaña de los parámetros que necesita en la "pregunta", tales como rango de horas, coordenadas y en qué formato se quiere la respuesta, y se hace la petición.

Una vez hecha la petición, se espera la respuesta y se saca la información deseada. Viene en formato de diccionario, así que es muy simple de filtrar la

información requerida, siendo en este caso los dos vectores de fechas y el de datos de temperatura. Ambos emparejados y localizados a la zona horaria española.

crear_df_tabla_temperaturas

Inmediatamente posterior a la obtención de fechas (y horas) y los datos de temperatura asociados, y siguiendo el mismo flujo que en los módulos anteriores, sigue la **creación del *DataFrame*** para poner los datos en el formato correcto, más la validación de dichos datos.

Teniendo los dos vectores, uno con fechas y horas, y otro con su dato asociado, se pueden reorganizar a un formato de tabla fácilmente como ya se ha visto en las funciones análogas en otros módulos. Se crea así el *DataFrame*.

Y también como en los módulos anteriores, es posible que los últimos datos estén incompletos debido a la cercanía de la fecha con el presente. Se optará por la misma solución que se implementó en el módulo de solares, referirse a *crear_df_tabla_irradancias* si se desea una explicación más completa. Pero resumen, se añaden al módulo dos funciones que serán la solución a ese problema, una de falseo de datos, y otra de purga de datos.

El de falseo de datos está pensado para el modo histórico, donde es aceptable tener algún dato puntual aproximado (más aún en la temperatura ya que no se usará en el modo de históricos). La purga se hace en el modo diario, que, ya que dentro del flujo del modo diario hay herramientas más avanzadas de predicción de datos, no es necesario hacer aproximaciones de datos, se pueden borrar y obtenerlos de dichas formas más precisas (nuevamente especialmente relevante en los datos de temperatura que la predicción será hecha por una web meteorológica especializada).

Una vez aplicada una de esas dos funciones se obtendrá un *DataFrame* válido, listo para guardar y/o usar.

falseo_datos

Función con el código exactamente igual a la función con la que comparte nombre en el módulo de Datos Solares. Se podría haber llamado a dicha función desde este módulo de temperaturas, pero se decide crear una "copia" en este módulo por simetría, y poder dejar autocontenido cada módulo. Referirse a dicha función de dicho módulo para una explicación más completa.

Como resumen rápido, es capaz de tomar un *DataFrame* entrante, y empezar a leer por el final. Iterará sus filas hasta que deje de encontrar filas con datos inválidos. Al encontrar una fila (día) completamente válido copiará su valor a los días que fueran inválidos, dando un *DataFrame* con **todos los días con datos correctos**, aún algún día no es perfectamente válido. Solo tiene sentido hacerlo si solo hay unos muy pocos días inválidos, manteniendo los errores bajos.

purga_datos

Tal y como en el caso anterior, esta función tiene exactamente el mismo código que su análoga en el módulo solar. Se hace una copia por las mismas razones que en la función anterior, simetría y mantener autocontenidos los módulos. Referirse a purga_datos del módulo de Datos Solares para más información.

Pero como resumen rápido, esta función está pensada para sustituir a la de falseo de datos en el modo diario. En este modo hay mejores herramientas para completar los datos faltantes o inválidos, así que **borra lo que no sea usable**, ya se completará mejor posteriormente.

guardar_temperaturas_csv

Siguiendo la estructura ya común en el resto de datos, le sigue el paso de **guardado de datos**.

Pero antes hay que decidir el nombre. El proceso es análogo y simétrico al resto de módulos nuevamente, basado en el estándar elegido se elige el nombre que tendrá el archivo a crear.

Una vez se tiene el nombre ya se guarda el archivo para su posterior uso. Se retorna la ruta del archivo.

datos_temperatura_df

Como ya se hizo para OMIE y Solares, los datos de **temperatura tienen su fuente automatizada**. Así que en el paso de intentar abrir y leer el archivo de datos creado con las funciones anteriores, si no existiera se podría volver a llamar estas mismas funciones para volver a crear.

Independientemente de si ya existía el archivo o se ha vuelto a crear, el resultado de esta función será el mismo, abrirá dicho archivo, lo leerá, y obtendrá un *DataFrame* (una tabla) con el que *Python* pueda trabajar cómodamente, y retornará dicho *DataFrame*.

buscar_datos_scrapeados

Otra función que comparte nombre con funciones análogas de otros módulos de datos. Y como ha sido el caso otras veces, esta función también compartirá estructura y lógica muy similar a dichas otras funciones. Referirse a los módulos de OMIE y de Solares para más información. Pero, en resumen:

Es una función exclusiva del modo diario, el cual completa con datos desde los obtenidos en el histórico hasta unos pocos días en el futuro. A diferencia de los otros datos, esta era una tarea imposible y en su lugar se hacía lo mejor posible. En el caso de las temperaturas gracias a la predicción de Open Meteo sí que será técnicamente posible esta empresa.

Pero aún no se hará eso. La tarea de esta función es optimizar este proceso de búsqueda. No habrá que obtener los datos dos veces si ya se tuviera alguno. Por tanto, antes de hacer esa búsqueda de datos futuros, se llama a esta función de **buscar** algunos de esos posibles **datos futuros ya existentes** de previas ejecuciones del código.

En el modo futuro se trabaja con parejas de archivos, uno de datos y otro de fuentes. Se mirará primero el archivo de fuentes para ver qué datos son reales. Se podrán recuperar estos datos y guardarlos apropiadamente, pues son estos los datos que busca esta función. También se mirará el rango de fechas con

fechas sin datos. Se obtiene así por tanto un *DataFrame* de datos de temperatura incompleto, así como un rango de fechas que se debe completar con datos aún. Se retornará este *DataFrame* incompleto de temperaturas y estas fechas que apuntan al paso de completarlo.

En cuanto a los archivos ya leídos se pueden borrar, el siguiente paso es crear unos archivos nuevos, más actualizados, que los sustituyan con la información obtenida aquí.

crear_nuevo_archivo_temperaturas_futuros

Tal y como se ha hecho en los módulos de datos anteriores, la función que le sigue a la obtención de posibles datos futuros previos es la **creación** en sí de los **archivos de datos futuros**. Y tal y como en los módulos anteriores su estructura será muy similar. La diferencia más destacable es que para temperaturas como ya se ha dicho se tiene un método alternativo para los datos futuros, pero para ese proceso existe una función dedicada distinta a la "normal" (*obtener_prediccion_temperaturas*). La estructura de esta función será análoga al resto solo que llamará a una función distinta para los datos.

Para más información referirse a sus funciones hermanas en los otros módulos de datos, pero resumiendo, en el modo futuro se trabaja con una pareja de archivos, una de datos y otra de fuentes. La de datos como implica su nombre serán los datos obtenidos de temperatura, y la de fuentes será de donde se obtuvieron los datos o que se deberá hacer con ellos. En este caso la fuente siempre será real así que el archivo de fuentes pierde su utilidad, pero se sigue haciendo así para mantener la simetría, ayuda en la depuración del código.

Se parte desde una base de datos históricos sumados a los posibles datos que se hayan podido obtener de activaciones previas del modo diario (función de *buscar_datos_scrapeados*) y un rango de fechas en el que generar los datos. En el caso de que no los tuviera o no llegaran a la fecha del presente, lo cual será el caso más común, se volverá a llamar la función de obtener datos históricos hasta el presente.

Ocurre aquí una situación complicada. La base de datos de históricos Open Meteo tarda un día o dos en actualizarse, y la predicción solo funciona para datos futuros. Aparecen por tanto uno o dos días con datos inválidos. En las otras funciones en el modo diario se ha solucionado simplemente purgando los datos y dejando que la IA los complete, pero la temperatura es la entrada de la IA, no

se puede usar la IA para completar la temperatura. Se toma así la decisión de falsear esos uno o dos días de datos incompletos entre la predicción y lo que contiene la base de datos de Open Meteo. No es lo más óptimo porque se busca que la temperatura sea lo más precisa posible, pero no es una imprecisión tan grave, se puede aceptar una temperatura ligeramente incorrecta por solo uno o dos días dentro del modo diario.

Con eso se tienen datos hasta el presente. Queda llamar a la función que se comunica con la API de Open Meteo para obtener datos futuros a partir de aquí para completar el *DataFrame* entero. Esta función es un poco diferente, no funciona por fechas. Sino por cantidad de días a futuro que se necesitan (hasta solo unos pocos días a futuro, pero más que suficiente para este problema). Se puede calcular fácilmente dicho número a partir de las fechas que ya se tienen.

Se obtienen así los datos de temperatura completos, pero en formato vectorial aún. Se pasan por la función de crear tabla de este mismo módulo para ponerlos en el formato correcto.

A partir de aquí vuelve a ser como en el resto de módulos. Ya que se tienen los datos, así que no hace falta rellenar con 0 nada. Sigue gestionar el *DataFrame* y futuro archivo de fuentes. Se asignarán aquí todas como reales, incluso las de la previsión. No son reales, pero no se van a poder mejorar. Como se ha dicho también, pierde ligeramente el sentido el archivo de fuentes aquí, pero es importante para la simetría.

Quedaría simplemente guardarlos en archivos y retornar las rutas de la pareja de datos y fuentes.

obtener_prediccion_temperaturas

Una función única de este módulo, y la que más lo diferencia de los otros. Se usa en el modo diario, y se comunica con la API de Open Meteo de *forecasting*, la cual permite **obtener datos de temperaturas futuras** por horas de muy buena calidad y muy fácilmente. Es una simple petición y la web da gratuitamente dichos datos.

En cuanto al código, esta es una petición a una API, así que hay que rellenar los campos que se requieran, los cuales son principalmente coordenadas y cuántos días se necesiten, además de en qué formato se quieren las temperaturas y las

fechas. Las coordenadas serán datos del problema, y en cuanto a la zona horaria, dato necesario para que las temperaturas estén alineadas con los datos de OMIE y eDistribución, también es un dato y es fácil localizar las horas. Restaría así la cantidad de datos que se requieren, que también es una tarea sencilla, simplemente restar la fecha inicial a la final y convertirlo a días (y sumarle 1 para el caso de borde, *edge case*).

Con estos datos ya se puede hacer la petición a la API. Devuelve los datos en un diccionario del que se pueden filtrar y obtener los datos precisos que se necesitan. Se devuelve así el vector de fechas (y horas, localizadas) y temperaturas futuras (predichas por Open Meteo), los cuales podrán sumarse a otros datos de temperatura existentes y serán unos datos muy importantes en la predicción de datos con IA de días futuros.

Emparejar Datos

El siguiente paso lógico luego de la inicialización de datos, tanto en el modo histórico como en el modo diario, es **cargar dichos datos y prepararlos para usarlos**.

Como módulo es sencillo, no hace nada “nuevo”, pero en funcionalidad es muy importante, es aquí donde se unifican todas las ramas de obtención de datos, todos distintos y con sus particularidades. Luego de este módulo saldrá una única variable con todos los datos emparejados y unificados.

Se guardará dicha variable en un archivo junto a los otros datos a modo de registro, no tendrá utilidad adicional aparte de depuración y comprobación de que los datos son correctos por el usuario.

Este módulo se llama desde el *main* (y alguna función concreta directamente desde el módulo de IA, pero son excepciones, más información en dicho módulo), desde la subrutina llamada *inicializar_vector_emparejados*.

Y análogo a los otros módulos documentados previamente, quedan líneas de código vestigiales al final de cuando se crearon las funciones, antes de estar implementado el *main*. Ya que dicho *main* está creado, y las funciones están implementadas, estas líneas al final han dejado de tener utilidad y han quedado desactualizadas. Se mantienen por depuración.

emparejar_datos_historicos

Tal y como en los módulos anteriores, existe una función que hace las veces ***main* del módulo,** para los datos **históricos** al menos. Es decir, la función que se llama desde el *main* real y gestiona todo lo necesario dentro del propio módulo, llamando las funciones en orden para mantener un nivel de abstracción alto en el *main* real y dejar los detalles autocontenidos en el módulo.

Concretamente, es la subrutina de traducción entre el *main* y el módulo, pero solo para los datos históricos (existe una similar para los datos futuros del modo diario).

Si se ha leído cómo funcionan los módulos de datos se puede intuir cuál será el primer paso de la función de emparejado. Ya que dichas funciones retornaban la ruta al archivo, pero no la información de dichos archivos en sí, el primer paso lógico es leer dichos archivos. Una lectura para cada archivo.

Se opta por crear un archivo, y luego leerlo en vez de directamente devolver los datos por robustez en el código y por eficiencia. Robustez porque el hecho de guardar archivos en el disco en vez de dejarlos cargados como variables internas le da “tangibilidad” a los datos, no se perderán si el código se interrumpe por cualquier motivo (desde pérdidas de suministro eléctrico hasta fallos computacionales, tanto *hardware* como *software*). Además, hay formas de retomar los datos desde lo que existiera ya en el disco previamente, así que no se pierde ese trabajo nunca. Y eficiencia porque, ya que el cargado de datos no es algo verdaderamente paralelo, sino un proceso secuencial, los datos se irían acumulando en memoria durante las diferentes subrutinas sin ningún aporte a dicha subrutina. El hecho de guardarlos y retomarlos cuando hagan falta libera memoria y poder de cálculo.

Pero cuando se van a usar los datos será necesario cargarlos, ya que el segundo paso será usarlos. Concretamente juntar los cuatro datos principales en un único archivo con la función *alinear_datos*, la función más importante de este módulo. Esta función es la que tomará las cuatro entradas y creará una única tabla (*DataFrame*) con todos los datos del problema.

Por precaución y robustez, ya que la función anterior es la más importante de este módulo, también es necesario asegurar que se ejecutó correctamente. Le sigue una función de comprobar paridad. Hace unas comprobaciones y se asegura que el resultado obtenido es el correcto. En caso contrario saltará un

error y el código parará automáticamente. No se puede empezar el cálculo si este *DataFrame* alineado tiene algún fallo. Sería un error extremadamente grave continuar, se arrastraría el fallo durante todo el cálculo dando un resultado totalmente errado.

Pero, por otro lado, si esta comprobación es correcta, ya se tiene una variable con todos los datos cargados y en el formato correcto, la cual será retornada.

load_endesa

Como ya se ha dicho en la función de emparejar los datos históricos, el primer paso es **recuperar los datos de los archivos** ya existentes.

Es una función con un código bastante simple, si existe el archivo se lee, y solo se retorna el rango de fechas que se pide según el archivo de configuración (una ejecución ininterrumpida y correcta del código debería devolver el archivo entero, no tiene sentido decidir no usar algunos datos, pero la funcionalidad se incluye y se le da al usuario la opción de decidir).

En el caso de que no existiera el archivo se dará un error, no hay forma de obtener los datos de demandas automáticamente (referirse al módulo de Datos Endesa para más información).

Retorna el *DataFrame* leído (los datos) en una variable.

load_omie

Conceptualmente idéntica a la función anterior. **Lee los datos del archivo** de la ruta proporcionada. Si existe, simplemente se lee. En el caso de que no existiera, para el caso de OMIE, y a diferencia de los datos de demanda, sí que se puede automatizar la obtención de los datos. Ya se ha definido una función en su respectivo módulo con esta capacidad, *datos_omie_df* del módulo del propio OMIE. Esta función además de crear el archivo para futuras referencias, devuelve ya también el *DataFrame* requerido para esta función de carga de datos, “ahorra” un paso de lectura del archivo (aquí no afecta a la eficiencia como

otras veces porque en este caso la creación del *DataFrame* va seguido inmediatamente por la utilización del mismo).

Con el *DataFrame* ya en una variable, se filtra por el rango requerido por el usuario como antes y se retorna la variable con los datos.

load_solar

Como ya es habitual en el código, esta función vuelve a ser simétrica a sus funciones hermanas, pero para los **datos de irradiancia**. Y como los datos solares también se pueden automatizar, su estructura y lógica es exactamente igual a la de load_omie, referirse a dicha función para más detalles.

Es decir, busca el archivo de la ruta proporcionada. Si existe lo lee sin más, y si no existe lanza una función del módulo solar, *datos_solar_df*, que no solo crea el archivo, sino que también devuelve el *DataFrame* deseado.

Finalmente filtra las fechas deseadas y retorna la variable con los datos.

load_temperatura

Nuevamente, función simétrica a las previas, pero con los **datos de temperatura**. Referirse a sus funciones análogas para más información.

Busca el archivo de la ruta proporcionada. Si existe lo lee en formato *DataFrame*. Si no existe lanza la función *datos_temperatura_df* del módulo de temperaturas y crea dicho archivo y además retorna el *DataFrame* requerido.

Filtra las fechas especificadas en configuración y retorna la variable de los datos.

alinear_datos

Como ya se ha indicado esta es la función más importante de este módulo. Es capaz de **tomar los datos cargados por las funciones previas y juntarlos todos** en una única variable, y además añadiendo en el proceso otra información útil para el cálculo y haciendo alguna transformación de los datos.

El primer detalle es que originalmente los datos solían venir en forma vectorial, pero para guardarlos se optó por darles una forma de tabla, más “humana”. Pero si se desea combinar los datos a no ser que se quiera usar una matriz en 3 dimensiones, es más útil la forma vectorial de los datos.

Es decir, se tienen varias matrices de dos dimensiones. Se cambiará sus formas a vectores (matrices de una dimensión) y se combinarán dichos vectores para formar otra matriz de dos dimensiones. Si no se hiciera este paso previo se combinarían matrices de dos dimensiones, dando una de tres. Esta matriz de 3 dimensiones (o incluso más) sería perfectamente usable por un ordenador, pero no tiene un formato “humano”, se hace más difícil la depuración y presentación de datos.

Python tiene herramientas para vectorizar una tabla, es un cambio sencillo. Lo que hay que tener en cuenta que estas tablas tenían 24 columnas (más la de la fecha), y un día por columna. Ahora se tendrán líneas de 24 filas por día (una por cada hora).

El siguiente paso es combinar estos vectores de datos pareja a pareja, teniendo cuidado de mantener el orden. Dicho orden será proporcionado por las fechas, pero como cada dato aporta su propia columna de fechas, se podrán borrar las fechas sobrantes, solo se necesita una al final. Las propias solo se usan de referencia para mantener el orden. Nuevamente *Python* tiene herramientas para hacer estas combinaciones por parejas, manteniendo el orden deseado y borrando las columnas indeseadas para la combinación final.

Y, lo que, es más, la salida de esta función del propio *Python* puede ser usada como una entrada para sí misma, así que se hace un proceso bastante sencillo el añadir columnas a la “tabla” que se está formando, se usan las mismas líneas de código para todas las combinaciones. Esta simpleza también es debido al hincapié que se ha hecho durante todo el código por hacer todas las funciones y datos simétricos, es una buena práctica que ha sido recompensada en este proceso.

Acabado este proceso se obtiene un *DataFrame* con las siguientes columnas: *DATE* (Fechas), Hora, Precio, Demanda, Irradiancia, Temperatura.

Este ya es un *DataFrame* usable, pero es mejorable. Primero, se pueden añadir algunas columnas más relacionadas a las fechas, como indicadores del día o el mes. Se añaden por tanto las siguientes columnas:

- ***Hora_int***: Originalmente se tenían las columnas nombradas del 1 al 24 según la hora del día. Se recupera ese concepto, pero solo con el número (es lo que se usará en el cálculo), es decir, se añade una columna que identifica con un número del 1 al 24 que hora del día es. Esto será útil para el cálculo de la IA, le ayudará a encontrar patrones. En cuanto a la implementación, nuevamente *Python* ya cuenta con herramientas para facilitar este proceso, referirse al código para ver la solución exacta.
- ***Dia_int***: También puede ser interesante tener numerados los días del 1 al N, siendo N el número total de días (un contador ascendente). No tiene muchos usos directos en el cálculo (solo en la IA se usa para dividir arbitrariamente qué cantidad de días serán para entrenar y cuáles para evaluar), pero es útil para el *debug*. Por ejemplo, un claro indicador de que algo ha ido mal es que el último número no coincida con el total de días, o que en algún punto se rompa la cadena de sumar uno a cada día que pasa, ya sea repitiendo número o saltando algunos.
- ***Dia_sem***: con un espíritu similar al primer parámetro, se añade una columna más que identifica en qué día de la semana se está, siendo 0 el identificador del lunes, y sumando uno por cada día, hasta llegar al 6, domingo. También lo usará en la IA, para encontrar patrones semanales.
- **Mes**: análogo a la variable previa, pero ahora con el mes, del 0 al 11, siendo 0 enero y 11 diciembre. Para que la IA pueda identificar patrones mensuales.

Ya que se han unificado todos los datos, se harán unos cálculos para unificar unidades y hacer los datos consistentes y coherentes entre sí, como cambio de escalas. Eso sí, hay que tener en cuenta que los cálculos solo se realizarán si el cálculo tiene sentido en primer lugar (solo se puede calcular con números, no con indicadores y *flags*). Esta función se reusará para los archivos de fuentes, los cuales no tienen datos numéricos. Se obviarán por tanto estos cálculos en su caso.

Ya que se están haciendo cálculos con los datos unificados, este es un buen punto para hacer la conversión entre irradiancia y potencia solar que ya se mencionó en el módulo solar. Se usa entonces la función *calculo_paneles* del módulo de solares (más información en su respectiva documentación). Esta función permite pasar del dato de irradiancia, dato que por sí solo no tiene mucha utilidad en este cálculo, a uno de potencias solares, el cual sí que es de mucha

más utilidad. Por tanto, se puede sustituir la columna original de Irradiancia por la recién calculada *PotenciaSolar*, pasando a ocupar su lugar en lo que queda de cálculo.

Queda así creado el *DataFrame* que se usará durante todo el cálculo, siendo este un hito importante en el cálculo. Ya se tiene una variable con todos los datos bien ordenados y listos para ser usados.

Como detalle, y para humanizar estos datos y facilitar la depuración del código se reordenan las columnas al siguiente orden:

DATE (fechas), *Dia_int* (contador), Mes (del 0 al 11), *Dia_sem* (del 0 al 6), Hora (H1 a H24), *Hora_int* (del 1 a 24), Precio (en euros el kwh), Demanda (en kwh), *PotenciaSolar* (en kwh), Temperatura (en grados Celsius).

Y por último se imprime en consola algunos datos, incluyendo la cabecera de los datos. Como se ha dicho es un hito importante, el hecho de que esto se imprima correctamente ya confirma que toda la parte de obtención y validación de datos se ha ejecutado correctamente.

comprobar_paridad

Como ya se ha dicho, la función de alinear datos representa un hito importante en el código, genera una función con todos los datos que se usarán durante el resto del código.

Surge naturalmente la necesidad de **comprobar que se haya creado correctamente esa variable**. Se comprueban varios aspectos generales de los datos que pueden indicar que haya habido algún error durante su creación. En el caso de que no pase una de estas validaciones se saldrá automáticamente a error y se parará abruptamente el código. Las comprobaciones que hace son:

- **Longitud**: la comprobación más básica posible, ver si tengo la misma cantidad de datos en todos los datos, tanto los de las fuentes como la longitud de la lista producto de la unión de dichas fuentes. Todas las longitudes deben ser iguales para pasar esta validación.
- **Cuenta de datos inválidos**: Busca todos los datos inválidos y los suma. Si dicha suma es mayor o igual a 1 no se cumplirá esta validación.
- **Detección de valores pico**: Hace una media de cada columna del *DataFrame*. Si hubiera algún valor 10 veces más alto que dicha media en alguna columna no se cumplirá esta validación. Sin embargo, si bien

extremadamente raro, podría ser posible que los datos varíen tanto. Esta validación no detendrá la ejecución del código, simplemente mandará por consola un aviso que instará al usuario a revisar los datos para buscar estos datos anormalmente alto y que decida manualmente si es correcto o no.

- **Revisión de la longitud de cada día**: Los días tienen 24 horas, ni más, ni menos. Y esto debería reflejarse en los datos. Reordena los datos para obtener los días y revisa que no haya ningún dato que tenga días distintos de 24 horas. Un solo día que no tenga sus 24 horas hará que falle la validación.

En el caso de que se cumplan todas las validaciones, se podrá retornar una confirmación de que el *DataFrame* se ha creado correctamente y se podrá continuar con el cálculo.

emparejar_datos_futuros

Análogo a la función de *emparejar_datos_futuros*, tendrá el mismo objetivo, pero con los datos futuros.

Hará las veces de ***main*** **el módulo** cuando se necesiten emparejar los datos **futuros**. Recordatorio que en los datos futuros se trabajan con parejas de archivos, uno de datos y otro de fuentes, se deberá emparejar ambos. La lógica es similar a dicha función análoga, referirse a ella para más detalles, se dará una versión resumida y puntuando las diferencias.

El primer paso es cargar archivos. En los datos futuros se trabajan con pares de datos y fuentes, y como hay 4 datos principales, se cargarán 8 archivos. Se reusarán las funciones de load previamente descritas, pero llamadas dos veces cada vez, una para cada archivo de la pareja.

Una vez cargados se ejecutará una función de alineamiento distinta a la comentada previamente *alinear_datos_futuros*, y, nuevamente, una vez a los datos y otras a las fuentes. Como detalle, aquí al procesar las fuentes se usará el modo de no ejecutar los cálculos que ya se dijo en su correspondiente apartado. No tiene sentido hacer cálculos con información no numérica.

Para acabar el proceso, y alineado con el resto de la función, se reusará la función de paridad también descrita en este módulo, solo que, ejecutada dos

veces, para datos y para fuentes. Como apunte, ahora no solo deben ser válidos las comprobaciones individuales que hace la función, pero además ambas ejecuciones de la función deben validar a la vez los datos obtenidos y devueltos, dando un grado de confianza mayor en los *DataFrames* creados.

Habiendo sido validados, se pueden guardar en archivos los *DataFrames* creados. Como en la versión del modo históricos, la única utilidad que tendrán estos archivos es de depuración, para ver que todo se está ejecutando correctamente y para tener un registro “tangible”. Los que usará el código son las variables creadas de datos y fuentes, las cuales serán las salidas de esta función.

alinear_datos_futuros

Como se ha comentado previamente, es una función especializada para **alinear los datos,** aplicada a los **datos futuros**.

Aunque en realidad no lo es, simplemente hace unos filtrados previos antes de ejecutar el alineado normal.

Este filtrado lo que hace es dejar acotados los datos futuros al rango de fechas especificado en el archivo de configuración, a una fecha mínima y máxima. Acota cada uno de los datos a estas fechas. Y serán estos datos acotados los que se alinearán con la función de *alinear_datos* original ya documentada en este módulo, retornando el mismo *DataFrame*.

alinear_datos_futuros_IA

Esta función es distinta a las vistas previamente en este módulo. Para empezar, es una función que ha quedado obsoleta y ya no se usa, queda como **vestigio**.

Y segundo, no estaba en el flujo de ejecución normal, esta función era usada para el entrenamiento y evaluación de los modelos de IA, un proceso que solo se hará una vez al configurar el código, y de forma personalizada y manual (más

información en el módulo de IA, en relación a la creación y el entrenamiento de los modelos de IA).

Se usaba en una etapa más temprana del código, y tenía como función obtener un *DataFrame* similar al que se obtiene con las funciones de alinear normales, pero enfocado a la temperatura, completando todas las columnas del *DataFrame* menos las de Precio, Demanda y *PotenciaSolar*. El objetivo era que las completara la IA.

Pero como se ha dicho, el entrenamiento de la IA ya no se usa así, se usan las funciones estándares del resto del código. Tanto así que usa subrutinas enteras del *main* fuera del propio *main*, es casi un tercer modo de ejecución, el modo de entrenamiento. Referirse al módulo de IA para más información.

Cálculo de la capacidad de la batería

Los módulos previos han sido la parte de procesamiento de datos, todos los pasos previos que hay que hacer para obtener y poner los datos en un formato usable. Y **el objetivo final de datos es el cálculo, el cual gestiona este módulo**.

No se puede decir que todo lo que se ha documentado hasta el momento, que suma apropiadamente un tercio de todo el código, sea inútil o que no aporta nada, pues es la base para el cálculo. Tal y como reza el lema de los Radiantes, viaje antes que destino, sin esos pasos previos no se puede iniciar el cálculo. Pero este módulo es el destino. O una parte de este al menos. Representa y materializa la idea y el objetivo de este código.

Una vez obtenidos los datos se van a hacer los cálculos. La versión más compleja de este cálculo es la del modo histórico, el cálculo del modo diario es trivial comparado con este, solo usa una parte de las funciones del cálculo.

La función principal de optimización tiene el objetivo de tomar los datos de entrada y calcular no solo la capacidad de batería óptima (mientras más grande sea la batería más cara será), sino que también el ciclo de demandas horario que deberá seguir la batería para maximizar el efecto del cambio de precio a diferentes horas del día, cargando a las horas más baratas y descargando en las horas punta, así como gestionando la carga obtenida por los paneles solares decidiendo en qué punto es el más correcto para hacer su descarga, de modo

que se evite lo más posible la compra de electricidad de la red en las horas punta. En resumen, controlar activamente la batería para reducir el consumo en horas punta y en su lugar desplazarlo hacia las horas valle, más baratas.

Este cálculo se debe hacer para cada día del que se tienen datos, y debido a la continuidad de los datos, también se debe mantener la correlación de demandas entre días. Puede parecer que el hecho de mantener las variables entre días puede ser un problema importante a resolver, pero en realidad no lo es. Planteando el problema correctamente ni siquiera aparece. Siendo ese planteamiento simplemente tomar todos los datos como un único vector, sin diferenciar entre días. Enfocado hacia ese planteamiento, ya se han organizado los datos de acorde a este planteamiento, siendo cada dato un vector de varios miles de datos.

Pero esto en sí mismo genera otro problema, el de hacer un **problema de optimización masivo de varios miles de datos** a la vez, sin tener la opción de dividir el problema en franjas horarias más pequeñas. Es por eso que se ha dedicado bastante tiempo y esfuerzo para implementar medidas para mejorar la eficiencia del cálculo, de modo que incluso pueda ser resuelto en unas pocas decenas de segundos incluso por procesadores poco potentes.

El resultado de este cálculo retornará no solo el ciclo de demandas de la batería, sino también la capacidad de batería óptima para el precio de mercado que se haya elegido, siendo el primer parámetro el más importante para el cálculo del modo diario, pero siendo el segundo el resultado preferido por el modo de históricos.

Aparece así naturalmente otra cuestión. Recordatorio, a más grande la batería más cara será, pero es el mercado el que marca cuánto se incrementará este precio. Si el mercado lo permite se podrá instalar una batería más grande manteniendo el coste lo suficientemente bajo. O viceversa. Se ha resuelto el problema para un precio concreto de precios de batería en el mercado, pero ¿qué pasaría si el precio de mercado cambia? Esta solución dejaría de ser válida, se debería recalcular el problema para obtener una nueva capacidad de batería óptima (también su respectivo ciclo de demandas, pero como ya se ha dicho esta solución será más importante en el modo diario, y en dicho modo ya estará fijada la capacidad de la batería, no va a afectarle el precio de mercado a su ciclo).

Para ahorrar tiempo y facilitarle la elección al usuario, **se calcula previamente una gran variedad de escenarios con distintos precios de mercado** de la batería, obteniendo para cada valor de precios una capacidad de batería óptima (y su ciclo). Estas parejas de datos llevan naturalmente a una gráfica, una con la capacidad de la batería óptima en el eje de ordenadas, y el precio de mercado

del kWh en el de abscisas. Esta es la información condensada de todo este cálculo, y se da al usuario una guía para en función del precio del kWh de la batería que pueda obtener en su situación actual, pueda saber qué batería es óptima en su caso, ni mayor ni menor.

Por supuesto el usuario es quien tiene la palabra final, pero esta es una guía matemáticamente objetiva que apoyará su decisión. Cabe destacar que, si bien la definición matemáticamente objetiva generalmente es positiva, puede también ser algo negativo. Puede ser que se calcule un valor matemáticamente válido pero impráctico en la realidad (especialmente relevante en los casos límites, se calcularán baterías enormes o minúsculas, según el extremo). Es aquí donde empieza el trabajo de la ingeniería, de usar las herramientas matemáticas para tomar decisiones realistas y prácticas. Por eso se dan dos gráficas, una general, con los valores matemáticamente correctos, y una segunda centrándose solo en los valores coherentes. Es la misma gráfica, pero “limpiando” los casos extremos no prácticos. Será esta segunda la que más valor e información le aporte al usuario, siendo la primera solo una “tendencia general”.

problema_rango_precios

Análogo a los módulos previos, hay una función que hace las veces de ***main*** **dentro del propio módulo**. Esta es dicha función para el modo de **históricos**.

Esta función será la encargada de plantear las múltiples iteraciones que deberán hacerse para obtener todos los escenarios posibles según varios precios de mercado del kWh de la batería. Tiene dos modos, un modo de Precio y otro de Capacidad. El modo de **Precio** es el modo principal, el cual toma un rango de precios, y definirá puntos en él, realizando un cálculo en cada punto. La densidad de puntos es un parámetro definido en el archivo de configuración, y podrá ser distinto según si se está en la gráfica general o en la de detalle. El modo de **Capacidad** es más directo, el vector en este caso se genera con capacidades de baterías, quedando fijadas dichas capacidades desde este punto. A partir de aquí se calculará usando estos valores de capacidad fijadas, obteniendo como dato relevante el ciclo. Ya puede intuirse con el dato obtenido que esta forma de cálculo es algo que no se hará en el modo histórico, pero en su lugar está más enfocado al diario. Solo que incluso ni siquiera en dicho modo se hace así actualmente. La opción de cálculo con una capacidad de batería fijada si bien es interesante, es una solución que usaba previamente, iterando por un vector. Ya no se usa este método, se optimizó el cálculo y se puede ser mucho más preciso para este cálculo. Queda a modo de vestigio, no se necesita iterar capacidades fijadas.

Una vez se tiene el vector sobre el cual se va a iterar, se puede plantear el cálculo. Se llamará a la función que resolverá el problema (*problema_optimizacion_historicos*), la cual hará un cálculo para cada valor del vector, dando como resultado una capacidad calculada para cada punto (el en modo Capacidad este resultado es redundante, pero por simetría y por usar las mismas estructuras).

Serán estas capacidades las que se retornarán, pero internamente durante el cálculo **se habrán ido guardando en dos archivos el cálculo completo** de cada punto en más detalle, siendo estos el verdadero resultado, los archivos creados.

problema_optimizacion_historicos

Función de **traducción entre la optimización y los datos seleccionados**. Pero esta, a diferencia de la anterior, entrará en mucho más detalle. Y menos el cálculo matemático de la optimización en sí, hará el proceso de entero, es la función de optimización principal.

El nivel de abstracción aquí baja significativamente, se usan y se trabajan variables de datos específicos. No es el objetivo de este documento bajar tanto la abstracción, por lo que es recomendable referirse al código para ver las soluciones exactas obtenidas. Se dará una versión conceptual de esta función en este documento para poder seguir el flujo del código.

Como se indicó previamente, el resultado real de este módulo serán los archivos que se crearán con los detalles del cálculo. Esta es la función que “rellenará” estos archivos (no los creará, como ya se indicó en el *main*, se usan bases de datos externas, y se explicó en dicho *main* también de cómo se crean y se borran).

La primera comprobación que hace es ver qué datos tiene. En el caso de que no tenga el precio unitario de la batería es que se realiza un cálculo único, el cual no procede del flujo normal de cálculo. Se asigna por tanto un valor sacado de la configuración.

El primer paso del cálculo es ver si se necesita calcular en primer lugar. Como se hacen dos cálculos, uno general y otro en detalle, es posible que ya se hayan calculado **algunos puntos** en el modo de generales que se puedan **reusar** para

el de detalle. Existe un archivo de datos indexados que “lista” los puntos calculados. Si encuentra el punto en esa lista puede recuperarlo de la base de datos de los cálculos generales y podrá obviar todo el cálculo, solo necesitará copiar el cálculo ya hecho en la base de datos de los detalles. En realidad, hace más comprobaciones y da más pasos, como por ejemplo comprobar la existencia de dichos archivos previamente y qué hacer en caso contrario, como intentar buscar otras alternativas de fuentes de datos. Pero como se ha indicado, esta es la versión simplificada.

En caso de que no exista, se procederá al cálculo. En la estructura del código hay un remanente de una comprobación de hardware para intentar hacer el cálculo en la GPU, más potente que una CPU, pero no tan disponible ni accesible. Sin embargo, como se ha destacado reiteradamente, se han implementado las suficientes mejoras de eficiencia del cálculo como para que el planteamiento del cálculo en la GPU deje de tener sentido. Incluso una CPU poco potente puede ejecutar este cálculo a un buen ritmo. Queda por tanto esta estructura de cálculo con GPU descartada y siempre se recurrirá a la CPU.

En cuanto al cálculo de la CPU este es ya el cálculo de optimización en sí, el cual tiene una función dedicada aún a más bajo nivel de abstracción que esta. Es la siguiente función que se documentará.

Una vez acabada la optimización, esta retornará una gran cantidad de datos. Sigue siendo objetivo de la función actual gestionar estos datos, al menos su orden. Se unificará todo el gestionado de datos, los cuales pueden venir de distintas fuentes y distintas formas. Se identificará qué forma tendrán esos datos y se pasarán a la función que gestionará el guardado de estos datos, *guardar_json_resultados,* del módulo de presentar datos.

En cuanto a la variable que se retornará, será la que espera la función previamente descrita, la cual espera un dato de capacidad óptima de la batería recién calculada. Se buscará esta capacidad de entre todas las posibles formas de los datos que pueda gestionar esta función, y la retornará.

calculo_CPU

Como ya se ha comentado en el apartado previo, esta es **la función que realiza la optimización en sí**. Toma de entrada todos los datos ya definidos y obtiene la capacidad de la batería óptima para un precio concreto (si se desea, se puede optar por pasar este parámetro como un dato en lugar de como incógnita).

También proporciona el uso que se dará a dicha batería, es decir, el ciclo hora a hora de esta que minimizará el precio de la factura eléctrica.

Tal como ocurrió en el apartado anterior, esta es una función con un grado de abstracción muy bajo. Aún menor que la previa. Se dará en esta documentación una versión más conceptual de la misma, si se desea ver la solución en detalle es recomendable ir al código, el cual está debidamente documentado también, pero a un nivel más técnico.

Antes de empezar a ver el flujo del código se debe introducir el optimizador. Como ya se indicó, este es un problema bastante complejo matemáticamente. Se requiere una herramienta matemática acorde. Por suerte, *Python* cuenta con una gran cantidad de estas herramientas en forma de librerías, cada una con sus ventajas e inconvenientes, mejores y peores en diferentes aplicaciones.

Se opta en este código por la librería ***cvxpy*** (CVXPY)**.** Sus primeras letras aluden a ***convex optimization*** y sus últimas a *Python*. Permite resolver problemas matemáticos que tienen el objetivo de maximizar o minimizar una variable objetivo, sujeta a varias condiciones y restricciones. Además, está estructurada de forma accesible al usuario, permitiendo definir esas restricciones como “declaraciones”, dejando al optimizador el cálculo y los tecnicismos. Con la definición de esta librería se puede deducir por qué fue elegida, resume casi perfectamente el problema que ha de resolverse en este código, y lo hace de una forma relativamente sencilla.

La librería *cvxpy* tiene 3 grandes bloques:

- **Variables**: Puede definir variables a optimizar. Estas no tendrán un valor numérico hasta el final que se resuelva el problema, y quedarán definidas con condiciones y relaciones con el resto del entorno del problema. Pueden interactuar con “números normales”, lo cual les da una gran flexibilidad. No hay un límite del número de variables que se pueden definir, y al resolver se le asignará un valor óptimo a cada una. Aunque a más variables más grados de libertad tendrá el optimizador, puede que no encuentre el resultado deseado si este grado es muy alto.
- **Condiciones**: Los requisitos que se deben cumplir, todas las restricciones que el optimizador debe tener en cuenta al proporcionar su respuesta. Son condiciones absolutas, no se pueden violar nunca. Esto lleva a que se debe ser cuidadoso al elegirlas, es posible que se definan algunas condiciones que sean contradictorias y haga que el cálculo falle porque no se encuentre solución. La naturaleza de estas condiciones es amplia, se pueden relacionar variables de optimización entre sí, o variables con números reales, incluyendo condiciones de igualdad absoluta o de mayor

o menor que. En el caso de trabajar con vectores también se pueden definir restricciones para valores específicos de dicho vector.

- ***Solver***: Resolver implica minimizar o maximizar una variable. Esto requiere una ecuación, la cual implique al menos una variable de optimización, y que, al resolverla, dé un valor concreto. Un número. El número que se minimizará o maximizará. En esencia la ecuación es una restricción más, se puede denominar la "restricción maestra". Una vez definida esta variable a optimizar, y añadidas las condiciones, se resuelve. La librería cuenta con varios *solvers* de código abierto, algoritmos matemáticos muy complejos, que toman el problema y buscan una solución. Habrá algoritmos más especializados para ciertos problemas, pero la librería es capaz de elegir automáticamente uno que funcione bien. El objetivo final y principal de este *solver* es dar el valor de la variable objetivo, maximizado o minimizado, lo que se haya indicado. Y como "resultado secundario" se podrá obtener también los valores de las variables implicadas en el cálculo que dan esa solución "principal".

Una vez se entiende cómo funciona la librería de cálculo, se puede **empezar a documentar el flujo de la optimización**. El primer paso es determinar si se requiere calcular la capacidad o es un parámetro fijo. Si es un valor fijo se asume que ya está la batería comprada y no influirá en el cálculo ni en la solución, se puede dejar su coste a 0. En el caso de que sea una incógnita entonces también se necesitará el precio unitario del kWh de batería (recordatorio, a mayor batería, mayor gasto inicial, pero más potencial de ahorro con el ciclo correcto). También cambiará la declaración de la variable. Si es un dato fijo será un "número normal" (un *int*, o *float* si tiene decimales). Pero si es un parámetro a calcular se habrá de definir el tipo de variable a uno a optimizar. Se mezclan dos tipos de datos muy distintos en esta variable de capacidad, pero es algo que el optimizador acepta y se adapta.

Le sigue la obtención del resto de datos que afectarán el cálculo, algunos datos de entrada de la función, otros vienen definidos en el archivo de configuración principal. Se hacen las conversiones necesarias, como, por ejemplo, la amortización queda definida con un dato en años. Para facilitar el cálculo se divide esta amortización en horas, y se podrá añadir a la ecuación directamente. Y no se pueden olvidar los cuatro datos principales del cálculo (aunque la temperatura no afecta aquí). También se definen algunas variables de optimización auxiliares, tal como la energía acumulada de la batería o los coeficientes solares. Esto es porque tras prueba y error, y con el objetivo de mejorar la eficiencia de este código, se llegó a la conclusión de que **este optimizador funciona mejor con varias variables de optimización, pero con pocas restricciones**. Este efecto es de órdenes de magnitud en la velocidad de cálculo. Se entrará en detalle sobre estas variables posteriormente.

Para recapitular, se hará un resumen de las variables principales:

- **Demanda de la casa (P_{casa}):** uno de los datos principales, datos de eDistribución, potencia horaria demandada por la casa.
- **Precio (Precio bat):** otro de los datos principales, con fuente en OMIE, precio horario de la potencia consumida.
- **Paneles (P_{solar}):** tercer dato principal, la potencia que generan los paneles solares.
- **Capacidad de la batería ($C_{\text{bat usable}}$) [si aplica]:** la variable de la que se habló previamente, que puede o no ser un parámetro a optimizar (en el modo históricos lo será). Capacidad usable de la batería (la máxima con un porcentaje elegido por el usuario).
- **Demanda de la batería ($P_{\text{batería}}$):** la variable más importante de este cálculo, la variable de optimización que una vez resuelta dará las directrices de cómo debe operar la batería para minimizar el costo, cargando en horas valle y descargando en horas pico. Potencia demandada de la batería.
- **Energía de la batería:** una variable auxiliar. Llevará el recuento de energía actual de la batería con un sumatorio acumulado de todas las potencias en el sistema. Tiene como objetivo hacer que nunca sea negativa la energía, ni que se pase de la capacidad calculada. No se usa en la ecuación de cálculo directamente.
- **Coeficientes solares ($\text{Coef}_{\text{solar}}$):** otra variable auxiliar. Tiene como objetivo simular la capacidad de desconexión a voluntad de los paneles solares, dándole la posibilidad al sistema de desconectar temporalmente la energía aportada por estos cuando supere la capacidad de consumo y acumulación de este. Si se tiene la opción de volcar y vender esta energía sobrante a red es una variable inútil y redundante, siempre estarán conectados. Pero no es algo que se pueda asumir tan fácilmente, especialmente con los desajustes que provocan en la red este vuelco de energía, dando lugar a potenciales apagones (Red Eléctrica de España (REE), 2025). Valor entre 0 y 1, siendo 1 usando toda la energía aportada, y 0 paneles solares desconectados.
- **Número de horas (Nº horas total):** según la garantía de la batería, o el tiempo en general que el usuario piense que puede mantener el sistema funcionando (parámetro del archivo de configuración), se puede amortizar el precio inicial de la batería en ese tiempo y sacar un coste añadido extra horario.
- **Horas de cálculo (Hora calc):** Vector de horas en las que se ejecuta el cálculo, para repartir e igualar este coste extra a la ecuación correctamente.

Con estas variables definidas, se puede plantear la ecuación a minimizar (Ecuación 2). El parámetro a minimizar será obviamente el coste, se puede definir como:

$$\text{Coste} = \sum \left[(P_{\text{casa}} + P_{\text{batería}} - P_{\text{solar}} \cdot \text{Coef}_{\text{solar}}) \cdot \text{Precio} \; + \; \frac{C_{\text{bat usable}} \cdot \text{Precio bat} \cdot \text{Hora calc}}{\text{Nº horas total}} \right]$$

Ecuación 2. Ecuación a optimizar (minimizar) implementada en el código.

Es decir, el coste es el sumatorio del balance de potencias total por el precio del valor de la electricidad en cada hora, y a eso se le añade la amortización horaria de la batería, un coste fijo dependiente del precio del kWh y de los propios kWh de la batería. Como detalle, al multiplicar los vectores del balance de potencias por los precios, se obtiene a su vez otro vector de costes por hora. A cada una de esas horas se le añade la amortización horaria, y después se suma todo.

Con eso queda finalizado el bloque de definiciones de variables, le sigue el de restricciones. Como ya se ha comentado, este solucionador funciona mejor con más variables y con pocas condiciones, pero muy potentes. Se opta por 6 conjuntos de condiciones:

- **Condiciones de energía final e inicial**: A través de la variable auxiliar de energía es sencillo, simplemente se necesita calcular la energía actual de la batería en función de las demandas. Se necesita hacer una suma acumulada de la potencia demandada de la batería. Es decir, se empieza a 0 (o al número que se quiera, esta es ya la condición de energía inicial), y se suma ese inicio a la primera demanda. A continuación, se suma ese resultado a la demanda de la segunda hora, y así hasta el final del vector. El último resultado de este vector que ha calculado será la energía final, aparece así la otra condición. Por defecto también será 0, pero se puede elegir cuál será. Es decir, **la energía acumulada es la integral de la demanda de la batería.**
- **Condición de pendiente de energía máxima y mínima:** La batería no puede actualizar su energía acumulada infinitamente rápido, hay pendientes tanto en la carga como en la descarga. Se necesita derivar (definición de pendiente) así la energía acumulada y hacer que no supere esos límites. O simplemente referir esta condición a la demanda de la batería, que como ya se ha dicho, la energía acumulada es la integral de la demanda. Lo que significa que lo contrario también es cierto, la demanda es la derivada de la energía acumulada. Gracias a esta variable auxiliar queda trivializada esta condición de pendiente máxima y mínima a una potencia de la batería máxima y mínima fijadas.
- **Condición de energía máxima y mínima:** También en relación a la energía acumulada, se necesitan definir los límites. El mínimo es simple, es 0. El máximo tampoco es mucho más difícil, solo se necesita determinar cuál es la capacidad usable que se está calculando (siendo la

usable la máxima con el porcentaje de profundidad de ciclo de la configuración aplicado). Y si ya existía previamente una batería, simplemente se añadirá a esta capacidad “gratis” (ya está comprada, no influye para el coste de amortización). La suma de estos dos valores es la energía máxima que se puede tener acumulada en cualquier momento.

- **Condición de potencia demandada máxima:** Tanto por contrato como por instalación eléctrica hay un límite de potencia contratada. El balance total de potencias no debe superar nunca ese límite, tanto en valores positivos como negativos. Los valores negativos representan la potencia inyectada en red. Técnicamente la potencia contratada no aplica en ese caso, ya que se está inyectando, no consumiendo, pero la instalación eléctrica sigue existiendo, y está dimensionada para un valor concreto. No se puede asegurar que soporte más de la potencia contratada, se limita entonces al mismo valor.
- **Condición de coeficiente solar:** para hacer al coeficiente solar un coeficiente propiamente dicho, hay que limitar sus valores entre 0 y 1. Más de 1 implican que es un multiplicador. Y menos de 0 no tiene sentido físico, sería decir que los paneles solares consumen energía.
- **Condición de inyección a red:** Como se ha dicho, se puede optar por inyectar o no a la red, decisión del usuario. Por defecto se opta por no hacerlo porque será la opción más “segura”, la que no depende de factores externos como la suministradora o la distribuidora permitan dicha inyección. Es simplemente permitir o no que el balance total de potencias pueda ser o no negativo.

Con las condiciones impuestas solo queda la última parte, resolver. Esta es la parte que menos intervención por parte del script, se puede delegar en la librería de optimización. Se iniciará un contador de tiempo por registro y herramienta para ver cuánto tarda en realizar el cálculo, y se resuelve.

Se obtienen del resultado no solo el coste minimizado, el resultado “principal”, pero también los resultados “secundarios” los que justifican el coste. Son estos resultados “secundarios” la definición de las variables definidas de optimización. También por comodidad y registro, se retornan el resto de variables del problema. En resumen: vector de precios de OMIE, vector de demandas de la casa de eDistribución, vector de energía solar generada (que no es igual a usada, para la usada se le aplican los coeficientes solares) de *Pysolar*, precio del kWh de la batería, **capacidad de la batería calculada**, capacidad usable de la batería calculada (el mismo que el anterior, pero solo un porcentaje), coste final optimizado, **vector de demanda de batería** (el ciclo optimizado buscado), vector de energía acumulada de la batería (integral del anterior), vector de coeficientes solares (consumo de la energía generada por los paneles solares).

Para finalizar se imprime por consola un resumen del cálculo (esos valores que se retornan, apoyándose en una función auxiliar), y adicionalmente el tiempo que

se ha tardado en calcular. También se retornan los valores requeridos, continuando la función de *problema_optimizacion_historicos* y dejando que dicha función lidie con los datos generados aquí.

preview_vector

Función auxiliar usada al final del cálculo. Lo único que hace es tomar el gran volumen de datos que genera esta función y los resume, humanizándolos para poderlos **imprimir por consola**.

comprobacion_hardware

Como ya se ha indicado, originalmente se pensó en implementar un modo de cálculo con CPU (procesador normal del ordenador) y otro con GPU (tarjeta gráfica), pero al final se acabó descartando esta idea por el buen rendimiento que ya se obtuvo de la CPU. Se mantiene esta función vestigial, que es la que hubiera decidido usar la GPU o CPU en función de si se tuvieran todos los **requisitos necesarios para usar la GPU**. La GPU necesita existir físicamente, además de estar instaladas las librerías de CUDA (NVIDIA) y *PyTorch* (PyTorch) para poder usarse. La CPU no tiene ninguno de estos requisitos.

Si bien esta función en este módulo es vestigial y ya no tiene uso, sí que se necesita una comprobación similar en el módulo de IA, allí si se usa la GPU y sí será necesaria esta verificación. Referirse a dicho módulo de IA para entender mejor cómo hubiera funcionado esta función.

Presentar Datos

Hasta el momento solo se ha ido generando información y presentando algún dato o mensaje puntual por consola como resumen y para ver que todo está funcionando correctamente y sin errores. Pero si se quieren usar los datos generados para algo se necesitan **guardar** correctamente y **presentarlos** de una forma fácil de entender. Es aquí donde entra este módulo. Es una colección

de funciones y de utilidades que se usa durante todo el código, disponible para que se usen y se llamen desde cualquier punto del código donde se necesiten. La mayoría de funciones son de representar gráficamente los datos obtenidos, pero hay alguna dedicada a guardar datos en u otras a leerlos si es una lectura más compleja.

Al igual que en el resto de módulos, existen al final del módulo unas líneas desactualizadas y sin uso de cuando se creó el módulo. Se mantienen por *debug*.

plot_simple

Una función bastante **simple** para **representar gráficamente**. Toma tres vectores de entrada, uno de ellos será el eje abscisas, el cual hará de base de los otros 2, que harán por descarte de ejes de ordenadas.

En este caso se opta por representar cada vector por una gráfica distinta, ya que tienen escalas y datos distintos.

Debido a su sencillez se puede intuir que es una función que se usó durante la creación del código, pero ya no se usa. Pero es también debido a su sencillez que se puede para presentar a la librería que se usara para hacer estas gráficas.

En *Python* existe una gran cantidad de librerías para este propósito, cada una con sus particularidades y enfoques. En el caso de este código no es necesario algo extremadamente complejo ni sofisticado, como se verá más adelante no se usan gráficas con funciones particularmente especiales, así que la librería ***matplotlib*** (Matplotlib) será la adecuada. Tiene muchas opciones, pero por lo general es una librería de propósito general sencilla de usar. Tiene la particularidad de que se le pueden ir añadiendo “opciones” y “condiciones” según se necesiten, así que según se necesiten funciones extras en la gráfica, se añaden líneas de código extras, como si fueran módulos adicionales.

plot_multiples

Gráfica que toma los **vectores de datos principales**, y algunos más relevantes, y los imprime por pantalla, **cada uno en su gráfica** distinta y dedicada (pero todas compartiendo el mismo eje horario en el de abscisas).

Parece un enunciado simple, pero cabe recordar que cada uno de esos vectores son datos horarios del periodo de cálculo, por lo que no es descabellado suponer que se estén intentando imprimir varios miles de valores por gráfico. Deja de ser un problema trivial.

La primera cuestión que aparece naturalmente es ¿cómo se representan gráficamente más de 1000 datos? Se podrían imprimir tal cual, simplemente 1000 puntos en la extensión horizontal de la pantalla. Pero eso comprime a los datos hasta el punto que simplemente no son visibles las diferencias diarias. Puede ser útil para tener una tendencia estacional, pero por el tipo de solución que resulta de una batería usable (relativamente pequeña) esta tendencia estacional o anual es prácticamente inexistente.

Por tanto, se necesita una forma de **segmentar** ese vector masivo en fragmentos más razonables. Se puede definir ese “fragmento razonable” como un día (24 horas). Se adaptará bien a la solución obtenida. Pero si se toma esa solución entonces aparecen varios cientos de esos fragmentos, es una mejora, pero no es suficiente aún, no es factible darle su propia gráfica a cada día. Más si se pretende imprimir varios vectores todos con el mismo problema. Así que, si se debe dividir en segmentos de un día, y solo se dispone de una gráfica para representarlos, se opta por la solución de representarlos todos unos encima de otros en la misma gráfica. Con esta solución se pierde la individualidad de cada día, pero a cambio se obtiene la tendencia general diaria de todo el problema, lo más representativo de estos datos. Además, si se deciden imprimir pocos días, se podrá distinguir entre dichos días, eliminándose el problema de la pérdida de individualidad.

Ahora bien, ¿cómo se divide el vector en segmentos diarios de forma que se pueda entrever la tendencia al menos? Con colores. Se define una lista de colores, y a **cada día se le asignará un color**. Se recorrerá esa lista cíclicamente ya que el caso más normal es que haya más segmentos que colores, habrá varios días por colores, pero es lo mejor que se puede hacer en este caso.

Entonces para cada gráfica con su respectivo dato se entrará en un bucle, en el cual se recorrerá dicha lista de colores y se imprimirá cada día sobre la misma gráfica, superponiendo valores a presentar, pero manteniéndose la tendencia

total. Además, se configuran varios parámetros adicionales con funciones estéticas, referirse al código para ver la solución exacta.

Y si bien se ha dicho que no tiene mucho sentido, la función es capaz de aceptar otros modos de segmentación, como el anual, por cuestiones de versatilidad y dar opciones para el código. Si se tiene una batería especialmente grande (sin sentido en esta aplicación), sí se podrán apreciar estos ciclos estacionales, estando cargando meses enteros cuando la electricidad es más barata y descargándolo en meses más caros. Pero nuevamente, esto es solo un resultado matemático, no uno coherente físicamente, se necesita una batería de cientos de kWh.

plot_multiples_aux

Es una función **auxiliar** con un funcionamiento prácticamente igual a la anterior. Se usa únicamente en el modo de depuración, y se usa en conjunto con la función anterior para poder ver representados gráficamente datos adicionales que normalmente quedarían internos y ocultos al usuario.

Además, ya que es una función distinta se pueden implementar pequeños cambios particulares que requieran dichos datos distintos. Es una herramienta más para el desarrollo, vestigial ya que finalizó dicho desarrollo.

plot_datos_por_dia.

Análogo al caso anterior, solo que esta función solo permite la impresión de **un solo vector**. Esta es de hecho la función base, desde la cual se amplió a más datos en dichas funciones anteriores.

Aunque esta función al ser solo un dato se puede añadir más información y detalle adicional, sin mencionar que al tener más espacio físico en pantalla en el eje vertical se ven mejor las variaciones de este. Pero esta particularidad es una herramienta para usos muy concretos y escasos. Otra herramienta del desarrollo, vestigio de este.

plot_guia_compra_doble.

Otra función de representar gráficamente datos, pero esta vez con uso y filosofía muy distintas. Esta vez solo tiene que imprimir dos simples vectores, y además sin grandes tendencias horarias o nada similar al caso anterior.

En papel parece una tarea sencilla, y lo es. Al menos en cuestión de representar gráficamente los datos. La dificultad radica en su objetivo. Esta es la función resultado final del **modo histórico**, la **función encargada de representar gráficamente la recomendación calculada** de qué capacidad de batería comprar. Una gráfica para la tendencia global, el cálculo matemático, el cálculo general, y otra para el rango usable en la realidad, el cálculo de detalle, con más puntos calculados y gráfica que verdaderamente será usada y referenciada.

Por tanto, ya se intuye que la verdadera dificultad será hacer llegar no solo al usuario, sino también al cliente la solución obtenida. Es posible que el usuario (alguien con al menos los conocimientos para ejecutar el código) y el cliente (alguien sin conocimientos técnicos) sean dos personas distintas. Se deberá presentar los datos de la forma más clara y estética posible. Se puede decir que es diseño gráfico.

En cuanto al código el primer paso será leer los resultados, que estarán guardados en dos archivos. También es posible que la función reciba los datos directamente desde variables ya con estos datos cargados. Se trataría de un proceso más simple entonces. A continuación, se crean las dos gráficas, una para datos generales y otra para datos detalle. Y se procede a introducir en la gráfica una serie de información adicional y aclaratoria, más toda una serie de detalles que facilitarán la lectura por parte del cliente. Nuevamente, referirse al código si se quieren saber qué “decisiones estéticas” exactas se han tomado.

guardar_json_resultados

Como ya se dijo al principio, este módulo no solo tenía funciones de representar gráficamente. Aquí empieza otra parte distinta en el módulo, una que **trabaja directamente con los datos y los guarda** para poder ser referenciados y usados posteriormente (por el mismo módulo de vuelta).

Primero, recordatorio de que en un inicio se usaban *json* para guardar los datos, pero debido a su lentitud se usa ahora una base de datos tipo *shelve* (Python Software Foundation). Se mantienen los dos métodos de entrada aun si el *json* ya no está en uso. Se carga la base de datos si se tiene, con el *json* como segunda opción.

Segundo, el dato a guardar puede venir de distintas fuentes. La función de cálculo ya identifica la fuente, pero es trabajo de esta función ejecutar un "protocolo" u otro según la forma de estos datos. Pudiendo estos datos ser salida de del cálculo directo, o bien ser un dato que ya existía en la base de datos. Se omitió el cálculo completamente en ese último caso, devolviendo el dato tal cual llegó.

En el primer caso se debe crear su **clave nueva**. Se asignan los valores que llegan de datos y se le da la forma de la base de datos (se define aquí). Además, se imprimen por pantalla los datos. Se imprimen como si fuera parte del cálculo. Este no es el módulo de cálculo, pero este guardado se ejecuta durante el cálculo. Para el usuario este proceso es transparente y continuo.

Para el segundo caso el paso de crear la clave y darle forma no es necesario, ya los datos vienen en el formato correcto, solo se guardan y se da por finalizado ese paso. Solo restaría imprimir para hacer el proceso transparente para el usuario como ya se hizo en el caso anterior.

Solo restaría cerrar la base de datos o *json*, según la fuente usada. Y además se indexa la clave al fichero de claves indexadas, para hacer más cómodo y rápido para el cálculo la comprobación de la existencia del cálculo ya hecho para no repetirlo. Simplemente comprueba si está en este archivo la clave, si no está se sigue con el cálculo ya que no se hizo previamente. Si lo encuentra entonces ya se tiene la información calculada, se simplifica el proceso.

leer_y_plot_json_resultados

Función de **leer los datos guardados y después representar gráficamente dichos datos leídos**. Se usa en las etapas finales del modo histórico. Esta es la función que se encarga de satisfacer la petición del usuario de que al acabar imprima la "justificación" para los de precios concretos elegidos.

Estos datos elegidos se encuentran en el archivo de configuración (referirse al apartado del *main*, y dentro de este al subapartado de *Parametros.json* para más información). Lo que da el modo histórico de normal es únicamente la gráfica de elección, que da una capacidad de batería recomendada para cada precio del kWh de la batería en el mercado calculado. Sin embargo, los ciclos calculados y los “detalles”, quedan ocultos. La justificación del dato representado gráficamente. Con estos parámetros el usuario indica no que los quiere ver, sino cuales ver. Existe una función dedicada para ver estos datos dentro de este módulo (*plot_multiples*). Pero esos datos están guardados. Es donde entra esta función, de leer y representar gráficamente.

El primer paso es leer los *json* de detalles y generales (para este punto ya se han volcado los datos de la base de datos en el *json*, no hay base de datos).

A continuación, se lee el fichero de configuración, leyendo los datos requeridos que especificó el usuario. Se buscarán los datos exactos en los *json* leídos (primero en los detalles, pues es más probable que esté ahí por volumen de datos). Pero como el usuario no sabe con exactitud qué datos se han calculado es muy probable que los datos requeridos no existan exactamente en los *json* de cálculo. Se da entonces el dato más cercano al requerido. Se dará un error máximo de medio paso de cálculo, un error lo suficientemente bueno. Se debe considerar también el caso de que no encuentre un valor ni exacto ni próximo, por algún motivo (potencialmente error de entrada de datos del usuario). Se omite esta petición y se continua con otro dato requerido entonces. Repitiéndose el mismo proceso hasta acabar con todas las peticiones del usuario.

Se imprimen los datos recuperados usando dicha función de *plot_multiples* para cada petición recuperada exitosamente del usuario. El código continúa una vez están impresas para seguir dichas gráficas con la gráfica principal y final de guía de compra, con los resultados de general y detalle que genera *plot_guia_compra_doble*.

guardar_json_para_ia

Función **auxiliar** similar a la función de *guardar_json_resultados*, pero dedicada exclusivamente al **entrenamiento de la IA**. Su función original era crear un set de datos para la IA y guardarlos en un *json* usando la misma filosofía usada previamente, pero durante el desarrollo se cambió la estructura.

Ya que el cálculo mejoró tanto no merece la pena pasar por este proceso de generación “dedicada” y guardado, para luego entrenar. No aporta mejora temporal y además es menos flexible al depender el entrenamiento de unos datos generados previamente, fijos.

Como ha sido el caso de otras funciones en el código, queda esta función como vestigio, sin uso real, pero con interés académico para ver la evolución del código.

carga_datos_temp_aux

Función **auxiliar para el entrenamiento de la IA** (vigente, en uso actualmente). Debido a la naturaleza de los datos de la IA, se requieren los datos de temperatura completos, pero el resto de datos no son “tan importantes” para el entrenamiento.

Se apoya entonces en las funciones del módulo encargado de emparejar datos y en los archivos que crea para poder obtener los datos propiamente generados y emparejados fuera del flujo de código normal.

Modelo de IA

Último módulo, y, como indica su nombre, **gestiona todo lo relacionado con la IA**. Desde la creación de los modelos hasta su uso en la predicción. Este módulo se usa **exclusivamente en el modo diario**, para completar y predecir datos. Y si bien su uso parece limitado, es el que más tiempo tomó desarrollar, tanto por los varios intentos que se tuvieron que hacer, a prueba y error con su aprendizaje en cada intento, como por el tiempo físico que toma entrenar un modelo de inteligencia artificial. Es difícil dar un total de tiempo, pero todas las pruebas ascienden a los cientos de horas de entrenamiento.

De hecho, este módulo no es el primer intento de IA que se tuvo en el código, cuando se mencionó el modo de GPU en el módulo de cálculo, una de las aproximaciones a ese problema fue una IA que aprendiera el patrón de demanda de la batería resuelta convencionalmente. Para eso se necesitaba calcular previamente dichos problemas con distintas variaciones de los datos de entrada,

lo que lo hacía muy poco conveniente y tedioso. Se mantienen los códigos vestigiales en dicho módulo, pero como se dijo, se descartó este enfoque. En parte porque el cálculo en la CPU mejoró sustancialmente, pero también porque no se consiguió un modelo que predijera correctamente la solución.

Lo que lleva a destacar la dificultad de trabajar con modelos de IA. Son una herramienta con mucho potencial, pero conllevan una cantidad de trabajo y recursos inmensos. El entrenamiento en las etapas finales, el *fine tunning* (Xia, y otros, 2024), fue un proceso casi artesanal, con cada paso tomando horas de espera.

En el resto de módulos siempre se han dejado unas líneas de código al final inútiles de cuando se creó el módulo. En este caso no, estas líneas son perfectamente funcionales, son las usadas para entrenar el modelo de IA. Sin embargo, su uso requiere conocimiento técnico del lenguaje *Python*, así como de desarrollo de inteligencias artificiales. El código está debidamente documentado, pero no es un proceso sencillo, ni está pensado ni diseñado para ser accedido por un usuario no técnico.

Vistas todas las dificultades que tiene trabajar con inteligencias artificiales y redes neuronales, se debe justificar brevemente su elección. Algo en lo que **destaca la IA** y hace particularmente bien es la **búsqueda de patrones** y la predicción de valores basados en dichos patrones (Bishop, 2006). Los modelos clásicos de predicción destacan en tendencias generales más que en patrones específicos. Es decir, una vez se suman los patrones individuales de muchas personas aparecen tendencias generales más estables y por tanto sencillas de predecir por un modelo clásico, problema que por ejemplo tiene que resolver Red Eléctrica de España diariamente para calcular la demanda necesaria para el país (Red Eléctrica de España (REE), 2025). Pero en el caso de este código no aparece esta tendencia general. Se trabajan con los patrones individuales, mucho más caóticos por naturaleza. Pero, aunque sea más caótico eso no significa que no siga un patrón. Es aquí donde entran los modelos de inteligencia artificial. Se le aportará al modelo un histórico de datos de unos días previos, con datos tales como la temperatura, día de la semana, mes del año etc., así como el propio dato a predecir en el mismo periodo histórico. Será el objetivo de la IA analizar estos datos, ver como se ha comportado el dato histórico a predecir los días previos en unas condiciones determinadas, inferir patrones, y sabiendo las condiciones el día actual, el día del que se precisa esta información, predecir unos valores horarios del dato objetivo usando los patrones inferidos.

En resumen, **entrenar y adaptar un modelo de IA usando los datos específicos del usuario**, que entienda bien la situación su concreta y genere datos personalizados. Pero por otro lado tendrá poco o nulo valor para otro

usuario, debiéndose entrenar un modelo personalizado de IA para cada escenario distinto.

Se seguirá el flujo de entrenamiento de los modelos de la IA para documentar este módulo, es decir, el de las líneas finales del código.

Entrenamiento de la IA

Python tiene una función que puede detectar cuando se ha ejecutado un módulo como *main*. Si esto ocurre, el código está configurado para cargarse en modo entrenamiento.

Lo primero que se realiza es cargar el *main* normal, ya que depende de subrutinas existentes en dicho módulo, como por ejemplo todo el proceso de la carga de datos. No es la práctica más correcta, pero se hace de esta forma para aislar estas líneas de creación de IA que no deberían ser accesibles por el usuario por defecto dentro del flujo estándar. Le sigue el uso de dicho *main*, el procedimiento habitual de carga de datos históricos. Ya que se va a entrenar la IA con los datos históricos, no es necesario obtener datos futuros (ni es posible sin el modelo en realidad). La evaluación de datos se hará con una parte reservada de dichos datos históricos.

A continuación, se introducen algunos parámetros “manualmente” que una activación del *main* habitual hubiera sido capaz de obtener automáticamente y se sigue con la predicción de temperaturas, del módulo de temperaturas. Es el único dato del que se necesitan datos “futuros”, y el único que no requiere la IA u otro método auxiliar para poder obtener.

Le sigue la definición de días que se desean reservar solo para evaluación del modelo, y unas variables con banderas que se deben editar manualmente en el código, las cuales indicarán qué tareas se ejecutan. Qué modelos se entrenarán y que modelos se evaluarán.

Y antes de empezar con la IA, se hace una predicción con un modelo clásico de predicción de series temporales, estilo a un modelo ARIMA (*AutoRegressive Integrated Moving Average*) (Box, y otros, 2015), pero simplificado. Se usará como base y punto de referencia para comparar el modelo entrenado.

Es ahora cuando se comienza con el entrenamiento y evaluación de cada modelo. **Un modelo por dato** (predicción de **consumo eléctrico**, de **precios de la electricidad**, y de **potencia solar** obtenida). Se leen aquí las banderas

definidas manualmente, y solo se ejecuta la tarea si las banderas se lo permiten. El entrenamiento tiene una función dedicada (*entrenar_ia*), la cual genera varios modelos en una carpeta. La evaluación usa los datos que se reservaron con este propósito, datos nuevos que el modelo no había visto nunca, y lo hace funcionar con estos nuevos casos. Calcula cuánto se desvió el resultado obtenido (método de mínimos cuadrados medios) con el dato real. La función de entrenar genera varios modelos intermedios, para probar cómo funcionan estos modelos se puede desactivar la *flag* de entrenamiento y dejar activa la de evaluación, cambiando el modelo que se evalúa introduciendo su ruta manualmente. **No siempre el mejor modelo será el que más entrenamiento tenga**, especialmente en las etapas finales, puede “aprender mal”, dándose casos de *overtrain* u *overfitting* (Zhang, y otros, 2017).

Se entra entonces en un bucle de cambiar parámetros, entrenar varios modelos, evaluar los modelos generados, elegir el que mejor rinda, y volver a afinar los parámetros para volver a repetir pasos. Para cada dato, con ajustes distintos, cada dato responde distinto. Y tomando cada bucle varias horas que dura el entrenamiento.

entrenar_ia

La primera y más importante función mencionada es la de entrenar la IA. Como su nombre indica es la que hace el **entrenamiento**. O más bien, llama a las funciones adecuadas, aún se está a un nivel de abstracción alto.

Define la carpeta y ruta, y llama a la función que prepara los datos para el entrenamiento (*preparar_datos_para_training*) y se los pasa a la función que hace el entrenamiento en sí (*entrenar_dual_input*). Y guarda el modelo.

preparar_datos_para_training

Primera función de este módulo en la que ya se entra a un bajo nivel de abstracción. No es objeto de este texto profundizar a tanto detalle, se refiere al lector al código, el cual está debidamente documentado, si se desea saber la solución exacta usada. Se mantendrá una explicación fácil de seguir en este documento.

Las inteligencias artificiales requieren **cantidades masivas de datos para su entrenamiento**, mientras mayor su calidad mejor. Actualmente se tienen dos años de datos históricos de estos datos de calidad, con cuatro datos principales (demanda del usuario, precios de la electricidad, potencia solar, temperatura), teniendo datos horarios de cada uno de estos. Esto son decenas de miles de datos, los cuales pueden parecer suficientes, pero no, no es ni por asomo una cantidad suficiente de datos. Se pueden usar incluso los datos auxiliares que se añaden al *DataFrame* principal generado en el módulo de emparejar datos para subir este número “artificialmente” a cientos de miles, pero este número sigue sin ser remotamente cercano al suficiente.

Se deben **generar más datos, obtener cantidad sacrificando para ello parte de la calidad**. La mejor calidad posible son los datos reales, pero se necesitan más. Una forma sencilla de obtener más datos a partir de los que ya se tienen es meter ruido, cambiando calidad por cantidad. Pero se meterá un ruido “controlado” para mantener la mayor calidad posible, datos que sean factibles en la realidad. El ruido no se aplicará a los datos aún, pero sí que se elige una “escala” de ruido aquí. Por prueba y error se ha encontrado qué nivel de ruido funciona mejor para cada dato.

También, se necesitan adaptar los datos que se generaron en el módulo de emparejar datos de su versión humana a algo que entienda mejor la IA. El concepto es el mismo, aportarle a la IA contexto de “cuándo” se obtuvieron esos datos, tanto de hora, día y mes, datos ya existentes en el *DataFrame* generado por dicho módulo de Emparejar Datos. Sin embargo, están en una escala que la IA puede malinterpretar, relacionando “magnitud” del parámetro temporal a la magnitud de la salida. Por ejemplo, un lunes tiene un 0 asignado en su día, y un viernes 4. Un humano sabe que la magnitud del día no significa nada, es solo un indicador. Pero la IA puede llegar a relacionar que a mayor este número, por ejemplo, mayor el resultado de salida, lo cual es incorrecto. Se deben traducir estos valores. La traducción en realidad es sencilla, se basa en pasar estos números a series de senos y cosenos. No solo se **regulariza** así su magnitud, sino que además por la forma en la que trabajan estas funciones trigonométricas ya llevan implícitas en sí una componente cíclica, temporal. Se necesitan ambos seno y coseno por cada dato para no perder información (hay dos valores que producen un 0 en el coseno, al añadir un seno se podrá inferir la información original viendo si este vale 1 o -1).

La temperatura no es un valor cíclico así que no tiene sentido aplicarle esta solución, pero aun así se debe normalizar. Se recurre a una solución más clásica, dividir su magnitud actual entre la magnitud máxima, dando un valor entre 0 y 1.

En cuanto al resto de los históricos no se deben normalizar, ya que la IA usará la propia escala de este valor para poder dar un valor análogo ya con la escala

correcta. Para ahorrar algunos recursos en memoria esta función seleccionará solo el histórico objetivo (demanda, precio o solar) y descartará el resto.

Por último, una vez traducidos los datos al formato que usa la IA, hay que traducirlos a su vez al formato que gráfica (*hardware*), el componente que hará el cálculo. La GPU tiene una potencia de cálculo enorme, pero solo si se puede paralelizar y adaptar el problema. Por suerte, existen las librerías de *PyTorch* (Python Software Foundation) y CUDA (NVIDIA), las cuales se encargan de estas cuestiones. Por parte del código solo se necesita cambiar los **vectores a tensores**, la variable especial que usa la gráfica. En esencia son matrices, pero muy optimizadas y especializadas.

Al final se genera un **bloque de datos** con varios miles de horas con la siguiente información: hora en forma coseno, hora en forma seno, día de la semana en forma coseno, día de la semana en forma seno, mes en forma coseno, mes en forma seno, temperatura normalizada, histórico objetivo (demanda, precio o solar, solo uno de los 3). Se retorna este bloque de datos junto a la escala del ruido que se ha decidido usar.

entrenar_dual_input

Nuevamente, otra función a muy bajo nivel, y una de las más complejas junto al modelo en sí. Se mantendrá la abstracción en este documento relativamente alta para hacerlo fácil de seguir. Tiene como objetivo **entrenar los modelos de IA**.

Esta función toma los datos ya preparados retornados por la función previa, tanto datos como el nivel de ruido usados. Si bien ya llevan un nivel de procesamiento alto aún no es suficiente para la IA. Se pasarán a un *DataSet* (*ForecastSingleFeatureDataset*). Más información en su respectivo subapartado de este módulo, abordado posteriormente. En resumen y viéndolo de momento como una caja negra, genera datos que la IA puede usar directamente para su entrenamiento. Le sigue un *DataLoader* genérico*,* que indica cómo se usan esos datos ya procesados dentro de la propia IA.

Por tanto, el paso inmediatamente posterior una vez obtenidos los datos es definir el modelo, la IA en sí, *DualInputForecastNet* (más información en su subapartado en este módulo). Con los datos y el modelo definidos y preparados, solo resta elegir un optimizador, el motor de cálculo que relaciona los dos. El elegido es Adam (Kingma, y otros, 2017), un optimizador de propósito general, suficiente para el propósito de este código.

Luego de unas comprobaciones generales, se definen las **condiciones y parámetros de entrenamiento**, así como la subfunción *loss_ponderada_asimetrica*. Esta parte de definir los parámetros es prácticamente artesanal. Es difícil documentar, pero, en resumen, estas condiciones son el control total del entrenamiento. Se puede ajustar individualmente la penalización tanto por "pasarse" de predicción como por "no llegar" (parte de asimétrica), así como definir rangos horarios en los que se hace más hincapié (parte de ponderada). Con esto se obtiene un parámetro de pérdida de cuánto se ha acercado la predicción del modelo a la realidad según le interese al usuario.

Con esta herramienta definida se empieza el entrenamiento, la parte más costosa computacionalmente del código. El entrenamiento consiste en "épocas" (*epoch*), siendo cada una de estas una variación de la anterior, efectuando cambios guiándose por el parámetro de pérdida generado previamente. Cada entrenamiento consiste en ejecutar estas épocas varios miles de veces, aplicando esta pérdida definida y avanzando el modelo, minimizando todo lo posible la pérdida según avanza el entrenamiento entre épocas. Cada cierto número de épocas se imprime por pantalla este parámetro de pérdida, y cada cierto otro número de épocas se hace una **copia de seguridad** del modelo. Estas copias son útiles por si se detiene el entrenamiento por cualquier motivo, como podría ser la pérdida del suministro eléctrico, o problemas *hardware* o *software*, entre otros. Se guarda así el progreso del entrenamiento y no se pierden horas de cálculo. Y ya que se imprime en pantalla la pérdida, se puede ver en qué momento se dio un caso de *overtraining* en las etapas finales, produciendo una de estas copias de seguridad intermedias un mejor resultado que el modelo final. Como se ha mencionado ya varias veces, este proceso implica una gran cantidad de implicación manual y prueba y error por parte del usuario. No es trivial.

Pero esta pérdida solo da una estimación general de lo bien que funciona el modelo con los datos en los que se entrenó. Se reservaron al inicio del código algunos datos que el modelo no ha visto nunca con este propósito, el de evaluación. Serán estos datos totalmente nuevos los que verificarán como actúa el modelo ante datos nuevos, si realmente ha "aprendido".

evaluar_modelo_con_df

Siguiendo las líneas de código al final del módulo, una vez terminado el entrenamiento y obtenidos varios modelos, le sigue la **evaluación** (si se permite con la bandera de evaluación para este dato concreto).

La evaluación es conceptualmente sencilla, simplemente **probar el modelo** recién entrenado en datos similares a los que fue entrenado pero que no vio nunca, **datos nuevos,** con el propósito de simular la realidad. Estos sin ruido aplicado, **aquí prima la calidad sobre la cantidad**, se quiere evaluar en el caso más realista posible. Se determinará con este proceso si el modelo realmente "aprendió" y pudo inferir los patrones, o si por el contrario solo "memorizó" los datos que tenía y es incapaz de actuar ante datos nuevos.

El primer paso es cargar el modelo a partir de su ruta. Confirmada su existencia se hará un proceso similar al entrenamiento, preparar los datos con *preparar_datos_para_training*, y luego pasarlos por el *DataSet*, *ForecastSingleFeatureDataset*, el cual genera un único bloque de datos que la IA podrá usar. Solo resta usar el *DataLoader* que le dirá al modelo cómo usar estos datos.

Se ejecuta el modelo con normalidad, se hace predecir unos datos basados en esta información nueva y reservada (no se entrena, solo se predice), y se comparan con los datos reales ya existentes en el histórico. Se retornan los datos tanto generados por el modelo como los datos reales.

Estos serán los datos que se manden a representar gráficamente en las últimas líneas del módulo, y **será el usuario el que decida manualmente si este es un modelo válido** y con potencial para seguir entrenando, o si por el contrario ha tomado una ruta incorrecta, quedando inútil y descartado. Como referencia, también se imprime por pantalla los resultados de un modelo clásico estilo ARIMA simplificado, se puede ver cómo rinde el modelo respecto a una solución tradicional. El objetivo es obtener un mejor resultado.

Nuevamente, deberá el usuario tener el criterio y el suficiente contexto e información de lo que pretende conseguir para poder interpretar y elegir los mejores modelos y resultados.

ForecastSingleFeatureDataset

Una vez introducidos los conceptos de *DataSet* y modelo, se pueden ver en más detalle, empezando por el *DataSet*. Simplificando, es la parte del código encargada de **tomar los datos cargados y adaptarlos al formato requerido por la IA**.

Para empezar, no es una “función normal”, es un **objeto** (Python Software Foundation). Sin entrar en detalles técnicos, esto es un método distinto a las funciones que incorpora *Python*, pudiendo definir varias funciones posibles y partes dentro del propio objeto, pero por simplicidad y claridad de la documentación, se tratará como una “función normal” con “apartados”.

La parte de ***init*** define las entradas que acepta, en este caso las salidas de la función de *preparar_datos_para_training*, un bloque de datos y una magnitud de la escala del ruido.

El apartado de ***len*** define la longitud y forma de los datos a retornar.

Y se llega así a la parte de ***getitem***, la parte donde más código se define. A esta parte llega un indicador *idx*, un parámetro que apunta a una zona de los datos. Se puede entrar en este punto en detalle de qué datos concretos recibe realmente la IA.

Llega todo un bloque masivo de datos, pero la IA solo usará el equivalente de unos pocos días del pasado (siendo el presente el indicador *idx*). Se debe recuperar de forma legible las variables del bloque de datos, pero solo los días requeridos.

También se gestiona aquí el **ruido para aumentar los datos artificialmente**. Se aplica aleatoriamente, solo el 20% de las veces se aplicará ruido. Y cuando se aplica se aplicará ruido con distribución normal, con la forma del parámetro a predecir. Se aplica el mismo ruido tanto a la temperatura como al parámetro a predecir. El 20% puede parecer poca probabilidad, pero nuevamente por prueba y error se encontró que este parámetro es el que mejor funcionaba. Más ruido baja demasiado la calidad de los datos. Además, el modelo de IA tiene opciones y funciones para aplicar “modificaciones internas aleatorias” para mitigar este problema de falta de datos, más detalle en el subapartado del modelo de la IA a continuación.

Dicho parámetro a predecir será el dato histórico real objetivo. Se tienen varios días de históricos. El último día no se usará en el entrenamiento en sí, pero será la referencia contra la cual se comparará lo generado por el modelo. Por tanto, este dato es ya una salida, solo que no la objetivo. El resto de datos históricos, con ruido aplicado se usarán en el entrenamiento como parte de los datos históricos, datos de referencia que el modelo puede consultar para obtener el dato siguiente.

El resto de parámetros, incluida la temperatura con el ruido aplicado formarán un bloque. Pero aparece un problema. Los datos históricos tienen un dato más que el dato del día a predecir. El propio parámetro objetivo. Es decir, los datos históricos cuentan con todos los datos que entran, pero para el día actual se ha descartado el dato objetivo (es el que hay que generar). Aparece una asimetría en las dimensiones de los datos actuales y los históricos. Una IA convencional de una única entrada y una salida no puede gestionar una asimetría de este tipo, así que se diseñará un modelo con una entrada dual. Una dedicada a los históricos, otra dedicada al día actual. Para reflejar esta asimetría se retornan los datos separados ya, en históricos (un bloque de varios días, con cada día incluyendo el dato objetivo) y en actuales, (una "fila" de solo un día, que además le falta el dato objetivo de actual, el que ha de predecir). También se retorna aparte el dato objetivo del día actual real. Como ya se ha dicho, el dato real no se usa en el entrenamiento del modelo, pero será la referencia contra la cual se compara lo generado por el modelo de IA.

DualInputForecastNet

Análogo al apartado anterior, el modelo de la IA tampoco es una función, sino un **objeto** con varios "apartados" internos. **Este modelo tomará los datos generados por el *DataSet*, y en conjunto con el *DataLoader,* generará unos datos de salida.**

Antes de entrar en el código en sí se debe conocer cómo funciona una IA. Sin entrar en detalles técnicos, un modelo de inteligencia artificial se compone de neuronas en capas. Algunas de estas neuronas y capas están especializadas para ser entradas, otras para ser salidas. Se generan conexiones entre estas capas de neuronas, y a partir de asignar valores concretos a estas neuronas y las relaciones entre ellas, además de gracias a complejos modelos y teorías matemáticas, más la avanzada tecnología hardware de las gráficas capaz de materializar estas teorías, se genera un valor a la salida (LeCun, y otros, 2015).

La red neuronal usada concretamente tiene tres partes. **Dos entradas y una salida**. Las partes de entrada (una para los datos históricos, otra para los datos actuales) tienen 3 capas. La capa de la entrada de datos actuales, solo tiene un día, y dicho día se compone de 7 datos. Aparecen así 7 canales de entrada unidimensional (solo un día). La capa de entrada de datos históricos deberá gestionar varios días a la vez, cada uno de 8 datos (los 7 del diario, más el dato objetivo histórico). Esta entrada será entonces de dos dimensiones (acepta varios días a la vez), y de 8 canales. Las capas de entrada y salidas tienen tantas neuronas como entradas, y las capas intermedias 256 neuronas. La capa que

une las dos primeras partes usa 512 neuronas (suma de las dimensiones de las predecesoras), seguida de una intermedia de 256 neuronas. La red finaliza con una única neurona, siendo esta la que genera la salida, un único canal, el objetivo actual. La Figura 24 representa gráficamente la topología de la red.

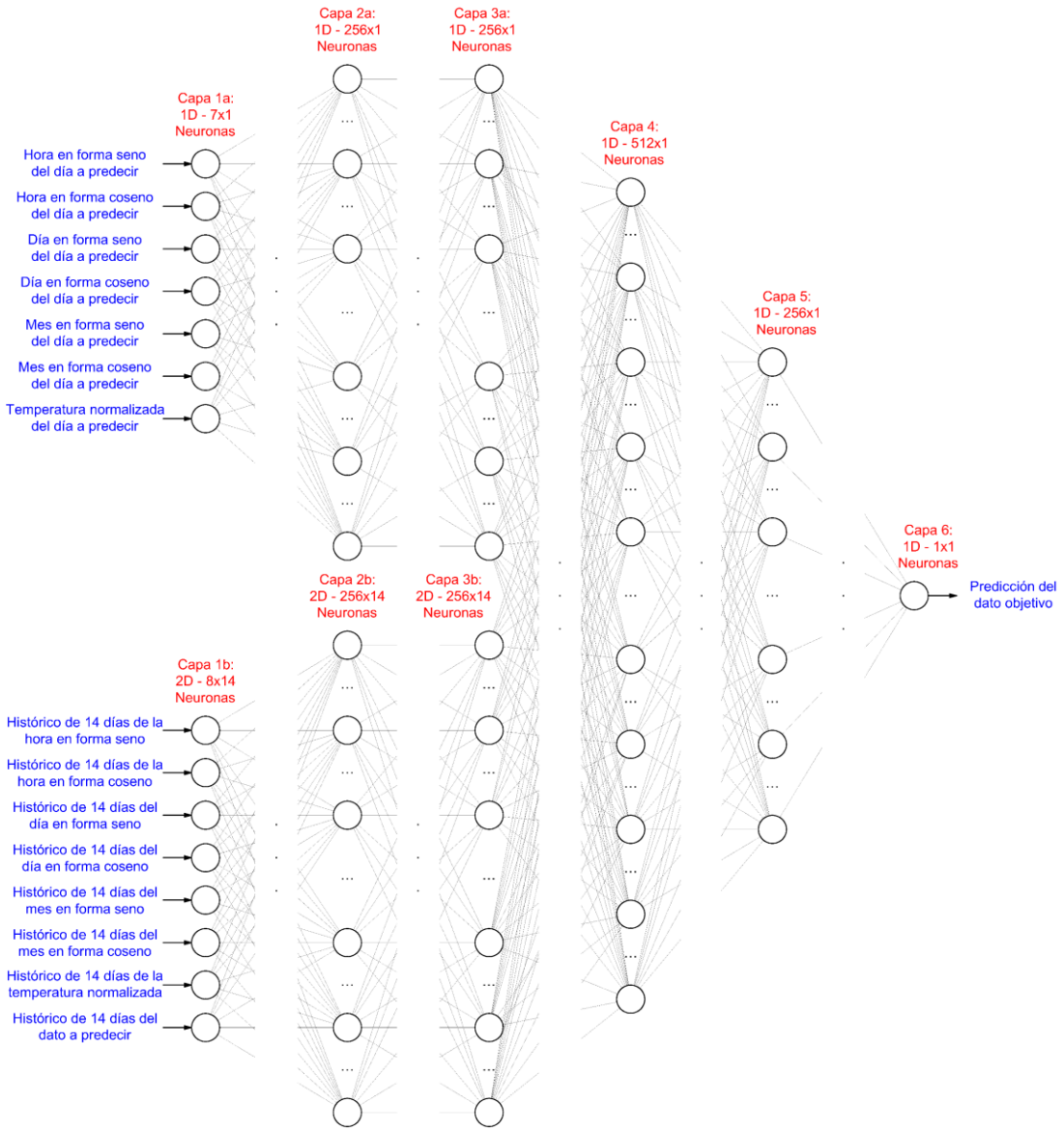


Figura 24. Topología de la Red Neuronal (modelo de IA).

Por suerte, la librería de *PyTorch* cuenta con todas las herramientas necesarias para la creación del modelo, se puede traducir este modelo a líneas de código relativamente fácil, tal y como se ve en el apartado de ***init***. Aunque con algunas líneas intermedias, como normalizaciones intermedias, y algún *DropOut* (Srivastava, y otros, 2014).

El ***DropOut*** es una técnica usada cuando se tienen pocos datos y se quiere evitar que la red neuronal los memorice durante el entrenamiento, dándose el fenómeno de *overfitting* previamente mencionado. Esta desactiva aleatoriamente algunas neuronas (20% en este caso) en cada activación durante el entrenamiento, de modo que el sistema gana robustez ya que no puede memorizar todos los casos del entrenamiento. En cada activación va aprendiendo a inferir patrones. Es por eso que es suficiente mantener los datos con ruido a solo el 20% tal y como se dijo durante el *DataSet*. Se mejora la calidad de los datos al no abusar del ruido. Al desactivar neuronas con parte de la solución memorizada, la red debe aprender a generar la solución sin memorizar los casos específicos.

En cuanto al apartado ***forward*** se puede ver la similitud respecto al *init*. En el *init* se inicializa y se define el modelo, pero es en el *forward* donde se usan estas capas de neuronas. Se aplican los mismos conceptos, no es necesario volver a repetirlos.

Finalmente **retorna un vector de datos**. Se le aplica una función de *ReLU*, un diodo lógico. Es decir, filtra y elimina los valores por debajo de 0, dejándolos a 0, sin modificar los valores positivos. Se hace este paso extra por seguridad durante el entrenamiento. Podrían aparecer aparecer potenciales errores, ya que no se espera que estos modelos generen datos negativos.

ResidualBlock1D

Se define una **capa especial para el modelo de IA**. También es un **objeto**, y su código es complejo, pero su idea no lo es tanto.

Lo que hace es crear dos capas de una dimensión cada una, con normalizaciones y *ReLU* (diodo lógico) intermedios para “controlar el caos” inherente a las capas de neuronas.

Para no complicar esta documentación más de lo necesario, todo lo que es necesario saber de este objeto es que está implementado en el modelo de IA, siguiendo la filosofía e ideas de este, referirse al código para su implementación exacta.

prediccion_matematica_horaria

Se ha hablado durante el módulo de un **modelo matemático más clásico**, un modelo ARIMA análogo a los que se usan en entornos más profesionales de predicción de datos, pero simplificado ya que no se dispone de las herramientas para implementar un modelo de tal complejidad. Es esta función la que lo implementa. Se usará de base para comparar cómo rinde el modelo de IA.

Como se puede suponer, si bien este modelo es más simple que un modelo de IA completo, sigue sin ser una tarea sencilla ya que está fundamentado en complejos modelos matemáticos. Por suerte, nuevamente *Python* tiene las librerías para ello. Se usará la librería ***statsmodels*** (statsmodels), concretamente su modelo OLS (*ordinary least squares*, similar a un ARIMA, pero simplificado).

El funcionamiento es en esencia similar al de la IA, se le aportarán datos de temperatura del pasado, así como datos históricos con esa temperatura, y a continuación se le pasará la temperatura actual y se esperará que el modelo matemático haya inferido una tendencia (que no patrón, esto es terreno de la IA) que le permita calcular el dato objetivo de hoy.

En cuanto al código, nuevamente se trabaja a un nivel de abstracción bajo, se recomienda referirse al código si se quiere conocer la solución exacta, pero, en resumen, se crean variables de la variable objetivo y de temperatura del pasado, una variable para cada uno de estos datos. Se pueden crear todas las variables necesarias. Por prueba y error se concluyó que este problema funciona mejor con tres días al pasado, por tanto, seis variables (tres de temperatura, tres del dato objetivo).

A continuación, se añaden los datos de temperatura del día de hoy. Con estos datos se carga un modelo de la librería de *statsmodels*. Como ya se ha dicho, el elegido es un OLS. Previamente, se debe afinar y calibrar el modelo con los datos. Internamente la librería es capaz de ajustar los coeficientes del modelo. Es un proceso bastante rápido, solo toma unos pocos segundos.

Con el modelo afinado se puede entonces predecir el dato buscado del día actual. Será uno de los datos retornados. También se retorna por comodidad el dato real, que vino en la función junto con el resto de datos históricos (recordatorio: aún no se está haciendo una predicción real, solo una evaluación, se tiene el dato objetivo real como un dato histórico). Además, se retorna el modelo OLS calibrado, por trazabilidad. No se usará, pero si se podrá imprimir por consola para ver los coeficientes y valores usados.

Esta predicción será una **predicción de referencia que la IA deberá superar**.

completar_datos

Una vez se ha creado unos **modelos funcionales de IA** llega el momento de **usarlos en el flujo normal de código**. Solo se usará en el **modo diario**.

Recordatorio, el modo diario luego de ejecutar la función *combinar_historicos_y_presentes* del propio *main* había quedado un *DataFrame* de datos y otro *DataFrame* con fuentes. Sin embargo, están incompletos. Tal y como apunta *DataFrame* de fuentes, muchos de los datos están marcados como "*AGenerar*", es decir, un modelo debe generarlos.

Es aquí donde entra esta función del módulo de IA. Lo primero que hace es ver qué recursos tiene disponibles, tanto de existencia de modelos de IA como de *hardware* (hay una función dedicada para comprobar el *hardware*, *comprobacion_hardware_y_modo*, más información en su subapartado).

Si se tiene un hardware correcto (una gráfica con las librerías de CUDA y *PyTorch instalados*), así como también los tres modelos de IA (para demanda, precio y solar) se podrá hacer la predicción usando la IA (debería ser el caso por defecto). En el caso de que no se cumpla alguna de esas condiciones se usará un modelo matemático más simple. No funciona tan bien, pero da un resultado aceptable.

Una vez se sabe qué modo se va a usar también se necesita saber qué datos se van a generar. Como ya se ha dicho esto es sencillo, el *DataFrame* de fuentes indica que datos han de generarse. Para cada dato se identifica el rango de fechas a generar.

Se entra así en funciones de *predecir_modelo_IA* o *predecir_modelo_clasico*, según la disponibilidad de hardware. Se comentarán más en detalle a continuación, pero, en resumen, estas funciones son capaces de tomar el *DataFrame* de datos completo, y basado en los datos que marcaba el de fuentes es capaz de modificar el propio *DataFrame* de datos, devolviéndolo con los datos que había que completar ya completos.

Se harán tres activaciones de la función, una para cada dato. Y como la función genera un *DataFrame* que solo ha actualizado los datos específicos que debía tocar, se puede usar la salida de una función como entrada de otra sin problemas, retornando así la última activación de esta función un ***DataFrame* de datos ya perfectamente funcional, listo para ser usado en el cálculo** de optimización del ciclo.

comprobacion_hardware_y_modo

Como ya se ha indicado en el apartado previo, esta función decide si se tienen los **recursos necesarios para usar la solución con IA**, o si por el contrario no es seguro.

La primera comprobación es la de la GPU, ver si se tiene una gráfica, y además también están instaladas las librerías de CUDA (de Nvidia) y de *PyTorch*. Para el entrenamiento es fundamental que exista la tarjeta gráfica, simplemente no se puede entrenar una inteligencia artificial sin una GPU. Sin embargo, para ejecutar el modelo no es tan crítico. Si se tiene la GPU se usará, pero en el caso de que no exista se podrá ejecutar los modelos en la CPU, aunque será más lento.

La segunda comprobación es si se tienen los modelos. Esta comprobación está enfocada al uso de dichos modelos. No se puede ejecutar un modelo de IA si no existe dicho modelo. Se recurrirá al modelo matemático si no existieran.

La función retorna dos banderas, cada una resultado de estas comprobaciones.

predecir_modelo_IA

Función que **completa los datos del *DataFrame*** principal marcados como incompletos usando un modelo de IA. Aunque en realidad esta función solo se encarga de completar el *DataFrame* sin tocar nada más. De los datos en sí se encarga una función dedicada a esto, *predecir_datos_df*.

Esta función primero hace algunas comprobaciones varias y alinea los datos para que la IA los pueda usar. Después llama a la función que trabaja con el modelo de IA para que genere un vector de datos predichos.

El siguiente paso es generar una máscara para poder aplicar estos datos generados sin tocar el resto de datos y dejar un *DataFrame* funcional a la salida.

Finaliza el proceso mandando un mensaje por consola, indicando que el proceso acabo correctamente. Retorna entonces el *DataFrame* con los datos de los valores objetivos completados con la IA.

En cuanto al *DataFrame* de fuentes no es necesario seguir actualizándolo, ya cumplió su objetivo diciéndole a esta función que datos se han de generar.

predecir_datos_df

La **función que interactúa con el modelo de IA para predecir los valores**. Nuevamente el nivel de abstracción en esta función es bajo, así que si se quiere saber la solución exacta usada se incita al lector a leer el código fuente, también con su documentación de guía.

El código es una versión simplificada de la función de *evaluar_modelo_con_df* porque en esencia necesita hacer lo mismo. Preparar datos con *DataSet*, cargarlos con el *DataLoader*, cargar el modelo de la IA y usar dicho modelo y dejar que prediga unos datos. La diferencia es que la función de evaluar comparaba los datos generados con los datos históricos. La función de predecir es el caso real, no puede comparar. No existen esos datos aún, así que ahí acabará la función. Estos datos predichos son ya el resultado final y los que se usarán en el cálculo de optimización. Cabe recalcar la importancia de la evaluación. Si no se ha hecho un buen trabajo creando el modelo los datos que

generará esta función de predecir datos puede que no sean lo suficientemente correctos.

En cuanto a código se dará una versión resumida de lo ya documentado en *evaluar_modelo_con_df*, pues el código es similar. Como ya se ha dicho, se cargan datos y modelo. Como particularidad, el *DataSet* usaba una escala de ruido para generar más datos. **Ya no es necesario ese ruido**, este es el **caso real**, simplemente se deja a 0 el ruido. Se usa por tanto una función especial para preparar estos datos, *preparar_datos_para_predecir_real*, se darán más detalles en el próximo apartado.

Se cargarán estos datos en el modelo y se ejecutara. Este proceso tardará más o menos en función de si se ejecutará en el procesador (CPU) o en la tarjeta gráfica (GPU), pero al cabo de unos segundos se tendrán los datos deseados. El modelo genera también unos “datos reales”. Esto es porque el modelo durante el entrenamiento le llegaban datos históricos reales, los cuales también retornaba con objetivo de evaluar su progreso. En este caso, en el caso real, esos datos “reales” son en realidad valores que estaban previamente marcados para generar por el modelo. No tienen un valor coherente, se pueden descartar. Solo se usarán los **datos predichos por el modelo**. Y serán estos datos predichos los datos deseados y retornados por la función.

preparar_datos_para_predecir_real

Esta es la función análoga a *preparar_datos_para_training* usada en la **ejecución real**. En realidad, su código es prácticamente igual, la única diferencia es que aquí no se aplicarán ruidos, se dirá que la escala de ruido sea 0.

Dicha asignación de ruido será la primera línea de la función. El resto de líneas serán análogas a la función original, referirse a la al apartado documentando la función de *preparar_datos_para_training* dentro de este propio módulo de IA para ver cómo funciona el resto del código.

predecir_modelo_clasico

Con las funciones de IA ya documentadas en su totalidad solo resta ver la opción de auxiliar, la solución usada en caso de que no existan modelos de IA y no se pueda ejecutar la función de *predecir_modelo_IA*.

Ya se ha comentado brevemente que ocurrirá, pero en resumen si la comprobación de hardware determina que **no es posible usar la IA** para la predicción real, **se usará entonces un modelo matemático clásico**. Concretamente un modelo ARIMA simplificado, un OLS. No será tan efectivo como un modelo de IA entrenado con los datos del usuario específicamente, pero el flujo del código debe seguir, y los datos predichos por este modelo son aceptables.

El código será muy similar al ya documentado en *prediccion_matematica_horaria*, ya que la solución usada será en esencia la misma. La diferencia es que esta función ya documentaba simplemente retornaba los datos sin más. Esta función al estar integrada en el flujo de código del modo diario deberá sustituir completamente a la función de *predecir_modelo_IA*, incluyendo dar una salida igual. Es decir, debe ver que rango de datos está marcado para completar dentro del *DataFrame* de datos original, y sustituir únicamente estos datos con los datos predichos por el modelo matemático, sin tocar nada más del *DataFrame*.

La parte de predicción de datos es equivalente a la función de *prediccion_matematica_horaria* como ya se ha mencionado. En resumen, buscará 3 días de históricos en el pasado, tanto del dato objetivo como de temperatura. Para el día actual solo se proporciona el dato de temperatura, esperando que genere dicho objetivo.

El modelo usado es un OSL de la librería *statsmodels*, el cual con estos datos históricos primero afinará dicho modelo, y con el modelo calibrado y con el dato actual de temperatura generará los datos objetivos del día de hoy, los datos deseados.

A continuación, debe modificar el *DataFrame* de datos según lo marcado como "*AGenerar*" por el *DataFrame* de fuentes. Se usará una solución análoga a la usada en *predecir_modelo_IA*, simplemente usar una máscara sobre el *DataFrame* de datos, modificando únicamente esos datos, y quedando preparado para ser retornado para continuar el flujo. Al igual que en esta función, no es necesario seguir gestionando el *DataFrame* de fuentes, ya que ha cumplido su objetivo indicando que datos del *DataFrame* de fuentes se debían editar y completar.

plot_dia

Función muy similar a la del módulo de presentar datos. Se podría usar una función de dicho módulo, pero se ha decidido crear una función aparte porque la IA tiene necesidades especiales que no tiene el resto del código. Esta es una función similar a la de *plot_simple* del módulo previamente mencionado, en caso de requerir más información referirse al apartado de dicho módulo.

Pero, en resumen, esta función toma **dos vectores, uno de demanda real y otro de demanda predicha** por un modelo (ya sea de IA o uno matemático clásico). Genera **dos gráficas**, una para cada vector. Además, añade algunas etiquetas simples, por claridad.

Es una función muy sencilla, que simplemente compara un vector de datos real con uno predicho. Pero por dicha sencillez no tiene mucho uso.

plot_multidia

Función en una situación parecida a la previamente documentada, una función muy similar a las del módulo de presentar datos (concretamente, esta es muy parecida a la de *plot_multidias* de dicho módulo). Como en el caso previo, se podría haber hecho uso de alguna de esas funciones, pero la IA tiene particularidades especiales que no encajan completamente con esas funciones, así que se ha creado esta función auxiliar dentro del propio módulo de IA.

Esta, a diferencia de la anterior resulta más compleja y avanzada, así que no se da el problema de tener poco uso por su simplicidad. Concretamente esta es la función que se encarga de **mostrar los resultados de la evaluación de los modelos**, mostrando gran cantidad de datos predichos por el modelo respecto a los reales.

Se ejecuta una vez para cada dato (si se ha activado la bandera de evaluación para dicho dato), y así como una ejecución adicional para la comparación con el modelo matemático a modo de base (esta comparación base ocurre siempre).

La función contiene tres gráficas. Una tiene información de los **datos predichos** por el modelo, la siguiente los **datos reales**, y la tercera y última una **resta de dichos datos**, para ver las diferencias. Idealmente deberían ser iguales, pero en caso de que haya discrepancias entre los datos predichos y los reales se verán en esta última gráfica las diferencias. Es útil para ver en qué puntos el modelo falla más y cuál es la tendencia de dicho fallo, para poder ajustar así los parámetros de entrenamiento en consecuencia.

Como todo el entrenamiento de modelos, no está diseñada para ser usada por usuarios no técnicos, así que se mantiene su “decoración” al mínimo, centrándose por encima de todo en funcionalidad.

En cuanto al código es muy similar a la función de *plot_multidias* del módulo de presentar datos como se ha mencionado previamente, así que en dicha función se ofrecece una explicación en más detalle, pero, en resumen, la función acepta una gran cantidad de datos, y para poder representarlos de forma fácil de entender para un humano se opta por segmentar dichos datos en días, superponiéndose los días uno encima de otro dentro de cada gráfica. Para poder distinguir los días se asignan colores distintos a cada uno. Pero como habrá más segmentos (días) que colores, estos colores se repetirán en bucle. Con esto se puede obtener la tendencia general por horas, aunque si se tienen pocos días podrán ser distinguidos individualmente gracias a los colores.