



**UNIVERSIDAD DE CASTILLA-LA MANCHA**  
**ESCUELA SUPERIOR DE INFORMÁTICA**

**GRADO EN INGENIERÍA INFORMÁTICA**  
**TECNOLOGÍA ESPECÍFICA DE COMPUTACIÓN**

**TRABAJO FIN DE GRADO**

Sistema Inteligente para la Gestión y Optimización de Energía  
basado en la Nube

Pablo Palomino Gómez

Julio, 2019





**UNIVERSIDAD DE CASTILLA-LA MANCHA  
ESCUELA SUPERIOR DE INFORMÁTICA**

**Departamento de Tecnologías y Sistemas de Información**

**TECNOLOGÍA ESPECÍFICA DE COMPUTACIÓN**

**TRABAJO FIN DE GRADO**

**Sistema Inteligente para la Gestión y Optimización de  
Energía basado en la Nube**

Autor(a): Pablo Palomino Gómez

Director(a): Luis Jiménez Linares

Director(a): Luis Rodríguez Benítez

Julio, 2019

Sistema Inteligente para la Gestión y Optimización de Energía basado en la Nube  
© Pablo Palomino Gómez, 2019

Este documento se distribuye con licencia Creative Commons Atribución Compartir Igual 4.0. El texto completo de la licencia puede obtenerse en <https://creativecommons.org/licenses/by-sa/4.0/>.

La copia y distribución de esta obra está permitida en todo el mundo, sin regalías y por cualquier medio, siempre que esta nota sea preservada. Se concede permiso para copiar y distribuir traducciones de este libro desde el español original a otro idioma, siempre que la traducción sea aprobada por el autor del libro y tanto el aviso de copyright como esta nota de permiso, sean preservados en todas las copias.



TRIBUNAL:

Presidente: \_\_\_\_\_

Vocal: \_\_\_\_\_

Secretario: \_\_\_\_\_

FECHA DE DEFENSA: \_\_\_\_\_

CALIFICACIÓN: \_\_\_\_\_

PRESIDENTE

VOCAL

SECRETARIO

Fdo.:

Fdo.:

Fdo.:



*A mis padres, por inculcarme el valor del esfuerzo*  
*A Nuria, por potenciarlo*





## **Resumen**

*(... versión del resumen en español ...)*

El resumen debe ocupar como máximo una página y en dicho espacio proporcionará información crucial sobre el *‘qué’* (problemática que trata de resolver el TFG), el *‘cómo’* (metodología para llegar a los resultados) y los objetivos alcanzados.



## **Abstract**

*(... english version of the abstract ...)*

Versión del resumen en inglés. En los trabajos cuyo idioma principal sea el inglés, el orden de Resumen y Abstract se invertirá.



# AGRADECIMIENTOS

---

Aunque es un apartado opcional, haremos bueno el refrán «*Es de bien nacidos, ser agradecidos*» si empleamos este espacio es un medio para agradecer a todos los que, de un modo u otro, han hecho posible que el TFG «llegue a buen puerto». Esta sección es ideal para agradecer a familiares, directores, profesores, compañeros, amigos, etc.

Estos agradecimientos pueden ser tan personales como se desee e incluir anécdotas y chascarrillos, pero nunca deberían ocupar más de una página.

*Pablo Palomino Gómez*



# ÍNDICE GENERAL

---

Índice de figuras	XVII
Índice de tablas	XIX
Índice de listados	XXI
Índice de algoritmos	XXIII
<b>1. Introducción</b>	<b>1</b>
<b>2. Objetivos e hipótesis de trabajo</b>	<b>3</b>
2.1. Objetivo Principal . . . . .	3
2.2. Objetivos Parciales . . . . .	4
<b>3. Metodología</b>	<b>7</b>
<b>4. Resultados</b>	<b>9</b>
4.1. Identificación y adquisición de las variables del sistema . . . . .	9
4.1.1. Variables de entrada . . . . .	9
4.1.2. Variables de salida . . . . .	12
4.1.3. Variables de control . . . . .	14
4.2. Aplicación de lógica difusa para la determinación de los estados meteorológicos . . . . .	15
4.2.1. Lógica difusa . . . . .	17
4.2.2. <i>Fuzzy Sets</i> . . . . .	17
4.3. Creación de las relaciones y restricciones propias del modelo . . . . .	20
4.3.1. Variables del PSR . . . . .	21
4.3.2. Restricciones del PSR . . . . .	22
4.3.3. Función Objetivo . . . . .	23
4.3.4. Implementación de la clase Simulation . . . . .	24
4.4. Generación optimizada de energía mediante programación lineal . . . . .	25
4.4.1. Programación Lineal . . . . .	25
4.4.2. Optimización con SciPy . . . . .	26
4.4.3. Implementación de las restricciones del tipo 1 . . . . .	28

4.4.4.	Implementación de las restricciones del tipo 2 . . . . .	29
4.4.5.	Implementación de las restricciones del tipo 3 . . . . .	29
4.4.6.	Implementación de las restricciones del tipo 4 . . . . .	30
4.4.7.	Implementación de las restricciones del tipo 5 . . . . .	30
4.4.8.	Generación optimizada de energía . . . . .	31
4.4.9.	Caso de prueba: Simulación del 11 de Marzo . . . . .	32
4.5.	Persistencia de datos y creación de la aplicación web . . . . .	34
4.5.1.	Persistencia con SQLAlchemy . . . . .	34
4.5.2.	Modelos User y Home . . . . .	35
4.5.3.	Implementación de una aplicación Flask . . . . .	38
4.5.4.	<i>Frontend</i> de la aplicación web . . . . .	43
4.6.	Migración de la aplicación a la nube mediante una <i>IaaS</i> . . . . .	45
<b>5.</b>	<b>Conclusiones</b>	<b>47</b>
<b>A.</b>	<b>Reporte de Simulación 11/04/2019</b>	<b>49</b>
<b>B.</b>	<b>Tests de la aplicación utilizando Nose</b>	<b>55</b>
	<b>Bibliografía</b>	<b>57</b>



# ÍNDICE DE FIGURAS

---

1.1. Dibujo del sistema . . . . .	2
2.1. Esquema del sistema . . . . .	3
4.1. Panel de control del área cliente Endesa . . . . .	13
4.2. Panel fotovoltaico de 50 W . . . . .	14
4.3. Batería estacionaria . . . . .	15
4.4. Lógica clásica vs lógica difusa . . . . .	17
4.5. Estados del cielo posibles AEMET . . . . .	19
4.6. <i>Fuzzy sets</i> de los estados meteorológicos . . . . .	20
4.7. Clase Simulation . . . . .	25
4.8. Esquema Cliente - Servidor . . . . .	38
4.9. Logo de eOptimizer . . . . .	39
4.10. Diagrama de flujo de eOptimizer . . . . .	40
4.11. Vista <i>dashboard</i> de la aplicación . . . . .	44
4.12. Vista <i>simulation report</i> de la aplicación . . . . .	44
4.13. Tabla <i>energy distribution</i> del 16/04 entre 13:00-23:00 . . . . .	45
4.14. Logo de la plataforma IBM Cloud . . . . .	46
4.15. Logo de Amazon Web Services . . . . .	46



# ÍNDICE DE TABLAS

---

4.1. Estados meteorológicos favorables . . . . .	20
4.2. <i>Endpoints</i> de la aplicación web . . . . .	41
B.1. Cobertura alcanzada en los tests del proyecto . . . . .	56



# ÍNDICE DE LISTADOS

---

4.1. Función para obtener los valores meteorológicos del día en curso . . . . .	10
4.2. Función para obtener los valores meteorológicos de un día concreto . . . . .	11
4.3. Ejemplo de respuesta de la API - AEMET para un día concreto . . . . .	11
4.4. Función para obtener los valores del precio eléctrico . . . . .	12
4.5. Ejemplo de respuesta de la API - AEMET para el día en curso . . . . .	16
4.6. Tipo de problema aplicable a Scipy.optimize.linprog . . . . .	27
4.7. Restricciones del tipo 1 . . . . .	28
4.8. Condición para dotar de valor los coeficientes . . . . .	28
4.9. Restricciones del tipo 2 . . . . .	29
4.10. Restricciones del tipo 3 . . . . .	30
4.11. Restricciones del tipo 4 . . . . .	30
4.12. Restricciones del tipo 5 . . . . .	31
4.13. Función de procesamiento del resultado a formato json . . . . .	32
4.14. Fichero de consumo por horas de Endesa . . . . .	33
4.15. Salida del algoritmo <i>linprog</i> . . . . .	34
4.16. Declaración de un modelo heredado de <i>Base</i> . . . . .	36
4.17. Modelo <i>User</i> . . . . .	36
4.18. Modelo <i>Home</i> . . . . .	37
4.19. Consulta para obtener el primer <i>User</i> . . . . .	38
4.20. Muestreo de errores en el formulario de inicio de sesión . . . . .	42
4.21. Módulo <i>run</i> para arrancar el servidor . . . . .	43
A.1. Reporte de simulación 11/04 . . . . .	49
B.1. Estructura del directorio <i>test</i> . . . . .	55



# ÍNDICE DE ALGORITMOS

---





# INTRODUCCIÓN

---

El consumo de energía juega un papel muy importante en el progreso y bienestar de la sociedad. Tanto es así, que la demanda energética no deja de crecer, por ello deben llevarse a cabo medidas para reducir el consumo elevado de energía, lo que se conoce como eficiencia energética. La eficiencia energética [17] se refiere al empleo de medios de optimización en la producción y aprovechamiento de la energía, con el objetivo de proteger el medio ambiente. Esto ha pasado a ser una necesidad debido a que las emisiones de  $CO_2$  van en aumento y el cambio climático es un hecho.

Por otro lado, puesto que las fuentes de energía fósil y nuclear son finitas, podría llegar el día en el que no se pueda satisfacer la demanda energética, salvo que se apueste por los métodos alternativos de obtención de energía. Es aquí donde entran en juego las energías renovables. Una de ellas es la energía solar [23], que permite el aprovechamiento de la radiación electromagnética del sol. Resulta interesante su estudio, debido a que es tan abundante que se considera inagotable: la cantidad de energía que el Sol vierte diariamente sobre la Tierra es diez mil veces mayor que la consumida al día en todo el planeta. Finalmente, además de ser una energía inagotable, es una energía limpia, una muy buena alternativa a los combustibles fósiles o energía nuclear.

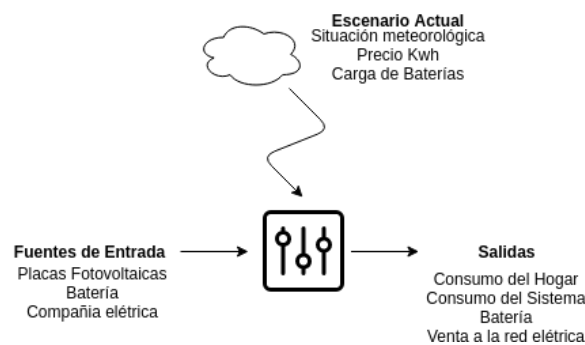
Teniendo en cuenta estos dos antecedentes, existe una motivación a la hora de obtener la energía demandada de la forma mas óptima y limpia posible. Además, también debe tenerse en cuenta el factor económico. Actualmente la mayoría de particulares tienen una única fuente de suministro de energía que vendría a ser la compañía eléctrica de la cuál son clientes, importando la totalidad de la energía que su hogar demanda a dicha compañía, a un precio establecido PVPC [4] (Precio voluntario al pequeño consumidor) que representa el precio máximo de referencia que pueden contratar los consumidores con hasta 10 Kwh de potencia contratada. Su valor tiene una discriminación horaria, lo que hace que en las horas de mayor consumo el precio sea mas alto. Sería interesante poder reducir la cantidad de energía que se obtiene de esta fuente en las horas pico (horas de máximo consumo donde el PVPC suele alcanzar el valor alto) y obtenerla de otra fuente cuyo precio sea menor, para así obtener un promedio mucho mas barato que con una única fuente de energía. Esto se puede lograr añadiendo nuevas fuentes al hogar, como puede ser la instalación de placas fotovoltaicas. Visto así parece muy sencillo y demasiado bueno para ser cierto, pero en el año 2015 mediante el Real Decreto 900/2015 [3] se establecieron unas condiciones para la instalación de placas fotovoltaicas, lo que se conoce coloquialmente como el "impuesto al sol". Por suerte, para potencias contratadas no superiores a 10 Kwh, no se pasa este impuesto, así que no es problema para el desarrollo y aplicación del sistema. Para tener el mínimo gasto posible, habría que obtener la cantidad óptima de cada una de las fuentes en cada momento, lo que supone un oficio tedioso y difícil para el ser humano. Afortunadamente para esta problemática, estamos inmersos en la era digital y las tecnologías de la información e inteligencia artificial están en auge. La inteligencia artificial [24] es la ciencia encargada de construir máquinas que:

- Piensan como humanos
- Piensan racionalmente

- Actúan como humanos
- Actúan racionalmente

Por lo tanto la inteligencia artificial podría abordar el problema definido anteriormente, mediante un sistema inteligente que se encargue de la gestión de energía automáticamente. Un sistema inteligente [24] es un sistema que aprende durante su existencia como actuar para alcanzar sus objetivos, en un entorno que le rodea.

Con todo lo expuesto antes, se puede concluir en que este trabajo se centrará en la creación de un **sistema inteligente** para la gestión de energía en el hogar de la manera más óptima y eficiente posible. En función de un escenario determinado en una hora  $t$  (situación meteorológica, precio del kilovatio-hora (kwh) en el mercado eléctrico, nivel de carga de las baterías de almacenaje, etc) se modelará la cantidad de energía eléctrica recibida por cada una de las entradas. De igual modo, se modelará la cantidad de energía eléctrica suministrada a cada una de las salidas. Esto se traduce en una optimización y aprovechamiento de la energía, que además tiene como consecuencia un ahorro económico en la obtención de la energía necesaria. En la Figura 1 se muestra un esquema del sistema donde se identifican desde un alto nivel de abstracción las entradas y salidas.



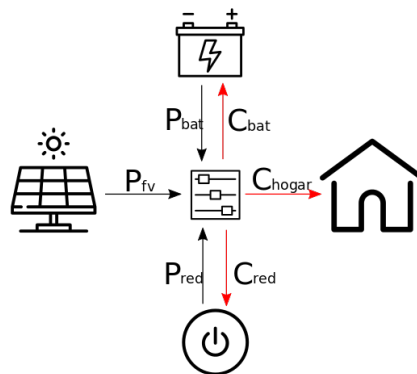
**Figura 1.1:** Dibujo del sistema

# OBJETIVOS E HIPÓTESIS DE TRABAJO

## 2.1. OBJETIVO PRINCIPAL

El objetivo principal a abordar será la construcción de un sistema inteligente para la simulación y predicción de la **distribución óptima** de energía entre **elementos generadores** y **elementos consumidores** en el hogar teniendo en cuenta que toda la energía generada debe ser consumida de una u otra forma.

El objetivo del sistema es ajustar en cada momento la cantidad de energía obtenida de cada elemento de los mencionados anteriormente con el objetivo de minimizar el gasto económico dedicado en el hogar. Este objetivo puede ser abordado como un problema de satisfacción de restricciones (PSR). Cómo se puede observar en la Figura 2, la funcionalidad del sistema sería la de modelar cada una de las salidas en función del valor de cada una de las entradas, que representan el conjunto de variables del problema de satisfacción de restricciones, el cuál se definirá mas adelante.



**Figura 2.1:** Esquema del sistema

En este sistema las fuentes de suministro de energía (entradas) son:

- Módulos fotovoltaicos
- Red eléctrica
- Baterías de almacenaje

Como fuentes de consumo (salidas) existen:

- Consumo energético del hogar
- Consumo propio del sistema que se propone
- Carga de baterías de almacenaje
- Venta al mercado eléctrico como particular

## 2.2. OBJETIVOS PARCIALES

A lo largo del trabajo habrá que satisfacer una serie de subobjetivos necesarios para lograr el objetivo principal tales como:

1. **Identificación y adquisición de los datos y variables que definen el sistema:** Estudio de las entradas del sistema y el grado de importancia que tiene cada una en cada situación. La información meteorológica será obtenida utilizando una API oficial de AEMET [9], y los datos del mercado eléctrico serán obtenidos utilizando la API oficial e-sios de Red Eléctrica de España S.A. [Esios]
2. **Establecer las relaciones y restricciones propias del consumo eléctrico:** Las variables obtenidas en el objetivo anterior estarán sujetas a unas restricciones que nos permitirán conocer las combinaciones posibles de valores, teniendo en cuenta que toda la energía generada debe ser consumida de alguna forma, ya sea por medio de venta a la red, o cargada en baterías de almacenaje en caso de superar la energía demandada por el consumo que se realiza.
3. **Añadir una IA para la generación optimizada de energía y dar lugar a una planificación:** Una vez obtenidos los datos y variables del problema y conociendo el grado de implicación de los mismos, se creará un modelo del sistema que dará una planificación temporal de 24 horas. Esto se podrá llevar a cabo incorporando inteligencia artificial.
4. **Simular la planificación:** Se llevará a cabo la simulación de la planificación realizando un seguimiento y una comprobación de la desviación que se puede haber producido con respecto a los datos reales. A partir de lo deducido en la simulación anterior, se tomará la decisión de ajustar o no el modelo para futuras predicciones.

Introduce y motiva la problemática (i.e. *¿cuál es el problema que se plantea y porqué es interesante su resolución?*)

Debe concretar y exponer detalladamente el problema a resolver, el entorno de trabajo, la situación y qué se pretende obtener. También puede contemplar las limitaciones y condicionantes a considerar para la resolución del problema (lenguaje de construcción, equipo físico, equipo lógico de base o de apoyo, etc.). Si se considera necesario, esta sección puede titularse *Objetivos del TFG e hipótesis de trabajo*. En este caso, se añadirán las hipótesis de trabajo que el alumno pretende demostrar con su TFG.

Una de las tareas más complicadas al proponer un TFG es plantear su Objetivo. La dificultad deriva de la falta de consenso respecto de lo que se entiende por *objetivo* de un trabajo de esta naturaleza. En primer lugar se debe distinguir entre dos tipos de objetivo:

1. La *finalidad específica* del TFG que se plantea para resolver una problemática concreta aplicando los métodos y herramientas adquiridos durante la formación académica. Por ejemplo, «*Desarrollo de una aplicación software para gestionar reservas hoteleras on-line*».
2. El *propósito académico* que la realización de un TFG tiene en la formación de un graduado. Por ejemplo, la *adquisición de competencias específicas de la especialización cursada*.

En el ámbito de la memoria del TFG se tiene que definir el primer tipo de objetivo, mientras que el segundo tipo de objetivo es el que se añade al elaborar la propuesta de un TFG presentada ante un comité para su aprobación. Este segundo tipo de objetivo no debe incluirse en el apartado correspondiente de la memoria y en todo caso puede valorarse su satisfacción en la sección de resultados y conclusiones.

Un objetivo bien planteado para el TFG debe estar determinado en términos del «*producto final*» esperado que resuelve un problema específico. Es por tanto un sustantivo que debería ser *concreto* y *medible*. El Objetivo planteado puede pertenecer una de las categorías que se indica a continuación:

- *Diseño y desarrollo de «artefactos»* (habitual en las ingenierías),
- *Estudio* que ofrece información novedosa sobre un tema (usual en las ramas de ciencias y humanidades), y
- *Validación de una hipótesis* de partida (propio de los trabajos científicos y menos habitual en el caso de los TFG).

Estas categorías no son excluyentes, de modo que es posible plantear un trabajo cuyo objetivo sea el diseño y desarrollo de un «artefacto» y éste implique un estudio previo o la validación de alguna hipótesis para guiar el proceso. En este caso y cuando el objetivo sea lo suficientemente amplio puede ser conveniente su descomposición en elementos más simples hablando de *subobjetivos*. Por ejemplo, un programa informático puede descomponerse en módulos o requerir un estudio previo para plantear un nuevo algoritmo que será preciso validar.

La descomposición de un objetivo principal en subobjetivos u objetivos secundarios debería ser natural (no forzada), bien justificada y sólo pertinente en los TFG de gran amplitud.

Junto con la definición del objetivo del TFG se puede especificar los *requisitos* que debe satisfacer la solución aportada. Estos requisitos especifican *características* que debe poseer la solución y *restricciones* que acotan su alcance. En el caso de TFG cuyo objetivo es el desarrollo de un «artefacto» los requisitos pueden ser *funcionales* y *no funcionales*.

Al redactar el objetivo de un TFG se debe evitar confundir los medios con el fin. Así es habitual encontrarse con objetivos definidos en términos de las *acciones* (verbos) o *tareas* que será preciso realizar para llegar al verdadero objetivo. Sin embargo, a la hora de planificar el desarrollo del trabajo si es apropiado descomponer todo el trabajo en *hitos* y estos en *tareas* para facilitar dicha *planificación*.

La categoría del objetivo planteado justifica modificaciones en la organización genérica de la memoria del TFG. Así en el caso de estudios y validación de hipótesis el apartado de resultados y conclusiones debería incluir los resultados de experimentación y los comentarios de cómo dichos resultados validan o refutan la hipótesis planteada.



---

## CAPÍTULO 3

# METODOLOGÍA

---

En el proyecto se emplea como metodología software un modelo iterativo e incremental, que da lugar a una toma de decisiones a corto plazo lo que se traduce en ampliar requisitos y soluciones en cada iteración (sprint) en función de las necesidades. Esto proporciona inmediatez y funcionalidad en el proyecto lo que hace que exista una mayor motivación e implicación en el mismo. Además, permite encontrar y solucionar errores a lo largo del trabajo y hace que el cliente esté más implicado debido a las numerosas entregas a lo largo del desarrollo del trabajo.

Como metodología de desarrollo y gestión de proyecto se usará **Extreme Programming (XP)**. La programación extrema [21] es un método ligero de desarrollo iterativo e incremental formulado por Kent Beck. Consta de varios períodos:

**Exploración:** Período donde el objetivo será identificar, priorizar y estimar los requisitos del trabajo, por lo tanto se obtendrá como salida un documento de especificación de requisitos. El cliente expone sus necesidades y los programadores deben eliminar la ambigüedad para asegurarse de que los objetivos pueden ser alcanzados.

**Punto de Fijación:** Se trata de una prueba rápida para profundizar en un determinado aspecto. Este punto se puede concretar durante la exploración o en cualquier otro momento en el que el equipo necesite resolver una cuestión.

**Planificación de la Versión:** Cada versión del sistema proporciona un valor de negocio al cliente, quien, en cada planificación de versión, selecciona las historias o requisitos que van a ser implementados. Esto proporciona el máximo valor de negocio aunque no sea lo más acertado técnicamente.

**Planificación de la Iteración:** Cada versión se divide en varias iteraciones. La longitud de iteración del trabajo se decide al principio y se mantiene constante durante el desarrollo. El equipo proporciona al cliente una estimación que representa cuanto trabajo se puede hacer en la iteración y el cliente selecciona que es lo que se implementará durante la iteración. Por lo tanto, se mantiene el marco de trabajo anteriormente mencionado en la planificación de la versión.

**Desarrollo:** El software se desarrolla para un caso de prueba. Cuando éste consigue satisfacerse se pasa al siguiente caso de prueba. Para integrar el código en el sistema principal se deben satisfacer todas las pruebas. Durante el desarrollo el equipo no debe intentar anticiparse a tareas futuras, solo centrarse en la tarea actual.





---

## CAPÍTULO 4

# RESULTADOS

---

En este capítulo se explican los resultados obtenidos en cada una de las iteraciones en el desarrollo de este trabajo fin de grado y cómo se relacionan con los objetivos definidos. Cada sección representa una iteración en el avance hacia el objetivo, donde a partir de la iteración 4 se realiza una integración continua hasta el hito final, pues dan como salida una versión funcional del objetivo.

### 4.1. IDENTIFICACIÓN Y ADQUISICIÓN DE LAS VARIABLES DEL SISTEMA

En la primera iteración se busca identificar cada una de las variables que entran en juego en el sistema, así como su proceso de obtención. Se debe tener clara la diferencia entre variables de entrada y salida y variables de control.

#### 4.1.1. Variables de entrada

Son las fuentes de suministro de energía al sistema. Habrá un total de tres:

- **Energía fotovoltaica (EF)**  
Energía procedente de las placas solares. Su valor viene determinado por varios factores, como son el número de módulos fotovoltaicos instalados y la máxima potencia posible en cada momento, *current module power* (CMP). Hace referencia a la potencia de salida, en vatios que produce un panel fotovoltaico en condiciones de máxima iluminación solar, con una radiación de aproximadamente 1 kW/m<sup>2</sup>. Será dependiente de la situación meteorológica del momento, de cuya obtención se hablará mas adelante. Como se puede observar, tendrá un valor máximo de obtención, que representa el tope de energía que podemos obtener de los módulos fotovoltaicos en ese momento.
- **Energía de red (ER)**  
Energía procedente de la compañía eléctrica como cliente particular. Al contrario que en el caso anterior, no existe un límite superior a la hora de obtener energía de esta fuente.
- **Energía almacenada en batería (EB)**  
Energía obtenida de la batería de almacenaje, que se ha guardado previamente para su posterior uso cuando el resto de fuentes de entrada tengan un mayor costo. Al igual que la energía fotovoltaica tiene un límite superior y viene determinado por la cantidad de carga de la misma y la profundidad de descarga que se le puede realizar sin resultar perjudicial para su ciclo de vida, que debe ser de un 50 % como máximo.

Cómo se ha mencionado, la energía fotovoltaica en una hora  $t$  será dependiente de la situación meteorológica de esa hora, algo evidente. Para contar con información meteorológica en desarrollo software existen conjuntos de herramientas o servicios que ponen dicha información a disposición del desarrollador,

como por ejemplo una API. Una API (*Application Programming Interface*) es un conjunto de reglas o especificaciones que permite a las aplicaciones proporcionar servicios a otras o comunicarse.

En este trabajo fin de grado se emplea la API oficial de AEMET [9]. Para su uso, se ha debido solicitar un **API key** ya que es una API cerrada, esto es, su uso está restringido a un conjunto cerrado de clientes. Para realizar una petición a la misma, debe incluirse el *API key* mencionado anteriormente en la url solicitada, así como una serie de parámetros como el código del municipio que se desea consultar.

Para obtener y procesar la información recibida por la API se ha creado el módulo *api\_aemet*, que contiene dos funciones para la obtención de información, una que se encarga de obtener la información referente al día en curso y otra que se encarga de obtener la información cuando la simulación se desea realizar de un día concreto. El motivo de diferenciarlas es que la API no realiza una respuesta con predicciones por horas de un día distinto al actual, devolviendo en su defecto un texto en lenguaje natural con un resumen de lo ocurrido meteorológicamente dicho día para estos casos. Dichas funciones se explican a continuación. Nótese que la url para la petición es obtenida como una constante de *const*, alias que hace referencia al módulo de constantes del proyecto: *project\_constants*.

- ***get\_weather\_today*** (listado 4.1). Esta función realiza una petición a la API mediante la librería *requests* [7], y en caso de obtener un código de éxito (código de estado http 200), procesa la respuesta recibida. Dicha respuesta es fácilmente procesable pues es un conjunto de parámetros clave-valor. Se procesa en la función ***create\_weather\_buffer*** y devuelve una lista con los 24 estados meteorológicos, correspondientes a las 24 horas de la simulación, del tipo: ["Despejado", "Poco Nuboso", "Despejado", ..., "Despejado"]

**Listado 4.1:** Función para obtener los valores meteorológicos del día en curso

```
1 def get_weather_today(city):
2     weather_buffer = []
3     url = const.AEMET_URL_NOW.replace('$CITY', city)
4     response = requests.get(url)
5     data = response.json()
6
7     if data['estado'] == 200:
8         url = data['datos']
9         response = requests.get(url)
10        data = response.json()[0]
11        weather_buffer = create_weather_buffer(data)
12    return weather_buffer
13 return None
```

- ***get\_weather\_archive*** (listado 4.2). Esta función realiza la petición a la API de manera similar a la anterior, salvo que debe incluir en la url el parámetro específico de la fecha que se desea consultar. En este caso la respuesta no es procesable tan fácilmente pues no se trata de parámetros clave-valor. En su lugar se ha de procesar un texto en lenguaje natural (Observese en el listado 4.1 cómo la respuesta es convertida a *json* y en este caso es convertida a *text*). Para su resolución, se ha implementado la función *process\_weather\_archive* que realiza una **búsqueda de ocurrencias** de estados meteorológicos conocidos en el texto en lenguaje natural, obteniendo así información acerca del estado meteorológico que se produjo ese día. Se forma el buffer con los estados meteorológicos encontrados en el texto y se devuelve como lista similar a la del caso anterior. En el listado 4.3 se muestra un ejemplo del tipo de respuesta obtenida. En este caso se encontraría una ocurrencia de un estado meteorológico ('despejado'), por lo tanto se formaría el buffer del estado del cielo con 'Despejado'.

**Listado 4.2:** Función para obtener los valores meteorológicos de un día concreto

```

1 def get_weather_archive(date, city):
2     weather_buffer = []
3     province = city[:2]
4     url = const.AEMET_URL_DATE.replace('$PROVINCE', ←
      ↪ province).replace('$DATE', date)
5
6     response = requests.get(url)
7     data = response.json()
8
9     if data['estado'] == 200:
10        url = data['datos']
11        response = requests.get(url)
12        raw_info = response.text
13        weather_buffer = process_weather_archive(raw_info)
14        return weather_buffer
15    else:
16        return None

```

**Listado 4.3:** Ejemplo de respuesta de la API - AEMET para un día concreto

```

AGENCIA ESTATAL DE METEOROLOGÍA

PREDICCIÓN PARA LA PROVINCIA DE TOLEDO
DÍA 12 DE FEBRERO DE 2019 A LAS 14:01 HORA OFICIAL
PREDICCIÓN VALIDA PARA EL MARTES 12

TOLEDO

Cielos despejados. Temperaturas mínimas en descenso. Temperaturas
máximas con pocos cambios predominando los aumentos en la Mancha.
Vientos flojos del este y nordeste tendiendo a flojos variables.

```

Las variables de entrada no son excluyentes, es decir, se puede obtener un tanto por ciento de la energía requerida procedente de cada una de ellas, lo que vendrá determinado por el precio en ese momento de cada una, ya que lo que buscamos es minimizar el gasto producido. A continuación se muestra el modo de determinación de los precios de las variables de entrada en una hora  $t$ :

El precio de la energía fotovoltaica se calcula a partir de la inversión realizada en la instalación de los módulos fotovoltaicos y la cantidad de años en los que se desea amortizar dicha inversión. Así, el precio en €/Kw de EF se toma a partir de la fórmula 4.1

$$Costo_{EF} = \frac{coste_{anual}}{promedio_{anual}^{kw}} \text{ €/kw} \quad (4.1)$$

Siendo el coste anual la cantidad invertida entre el número de años( $n$ ) en amortizarla, cómo se puede observar en la fórmula 4.2

$$Coste_{anual} = \frac{inversion}{n} \text{ €} \quad (4.2)$$

El precio de la energía de red es el ya comentado PVPC. Para obtenerlo, se hace uso de la **API oficial de Red Eléctrica de España (e-sios)** [5]. Para su uso se ha debido solicitar un *Token* de acceso que se utiliza en las llamadas a la misma, al tratarse de una API cerrada análogamente al caso de la API de AEMET. Para el procesado de esta API se ha creado el módulo *api\_esios*, que contiene la función *get\_incoming\_prices*, la cuál se muestra en el listado 4.4

**Listado 4.4:** Función para obtener los valores del precio eléctrico

```

1  def get_incoming_prices(indicator, start, end):
2      url = const.ESIOS_URL.replace('$INDICATOR', indicator)
3      url = url.replace('$START_DATE', dt.datetime.strftime(start, ↵
4          ↵ '%Y/%m/%d'))
5      url = url.replace('$END_DATE', dt.datetime.strftime(end, ↵
6          ↵ '%Y/%m/%d'))
7
8      response = requests.get(url, headers=HEADERS)
9      if response.status_code == 200:
10         data = response.json()
11         price_buffer = create_price_buffer(data, start)
12         return price_buffer
13     return None

```

Esta función es llamada desde el proyecto con el indicador deseado, que se corresponde con el precio que se desea consultar (en este caso PVPC), cuyo código numérico es obtenido de las constantes del proyecto, al igual que la url necesaria para la petición (ESIOS\_URL), que se forma con los parámetros adecuados y se realiza la petición *get* haciendo uso de la librería *requests* [7]. En este caso el *API key* no se concatena en la url, si no que debe incluirse en la cabecera de la petición en un campo específico, ya que se trata de una autenticación por token como tal. Si la petición ha sido exitosa (código de respuesta http 200), la función retornará un *buffer* de tamaño 24, que se corresponde con los valores del PVPC en las 24 horas definidas en la simulación. Para ello llama a *create\_price\_buffer*, que se encargará de generar la lista con los 24 valores del precio solicitado procesando la respuesta recibida de la petición a la API. De esta manera se consigue el precio por Kw de la energía de red en cada momento.

Por último, el precio de la energía de baterías se puede calcular de un modo muy parecido al de la energía fotovoltaica. Hace referencia al coste que supone extraer energía almacenada en la batería y está relacionado con la inversión realizada en la batería 4.3

$$\text{Costo}_{EB} = \frac{\text{coste}_{anual}}{\text{capacidad}_{bat} \cdot 182,5} \text{ €/kw} \quad (4.3)$$

Habiendo obtenido previamente el coste anual de forma similar a la fórmula 4.2. La constante 182,5 hace referencia al número de días del año (365) multiplicado por 0.5, debido a que no se va a realizar una profundidad de descarga mayor al 50 % de la capacidad total de la batería. El valor obtenido es el precio que supone recoger 1 Kw de la batería.

#### 4.1.2. Variables de salida

Representan las fuentes de consumo de energía del sistema. Habrá un total de cuatro:

- **Consumo del hogar (C)**

Demanda energética del hogar en cuestión, cuantía que debe ser satisfecha siempre, ya que es la energía que necesita el hogar para su uso cotidiano. En este trabajo fin de grado se ha decidido trabajar con clientes de la empresa eléctrica Endesa S.A., pues el hogar del alumno es su cliente lo que permitirá trabajar con información real. Además cuenta con un área privada de cliente que permite acceder a datos analíticos del hogar (Figura 4.1) y permite descargar ficheros en formato de texto con el consumo por horas de un día determinado en el hogar del cliente, justo lo que se necesita para dotar de valor esta variable. Para adaptar la información del fichero al valor de la variable se ha creado la función *read\_from\_file(filename)* del módulo *client\_consumption* que devuelve una lista con los 24 consumos de las 24 horas de la simulación en Kilowatios, obtenidos del fichero proporcionado.

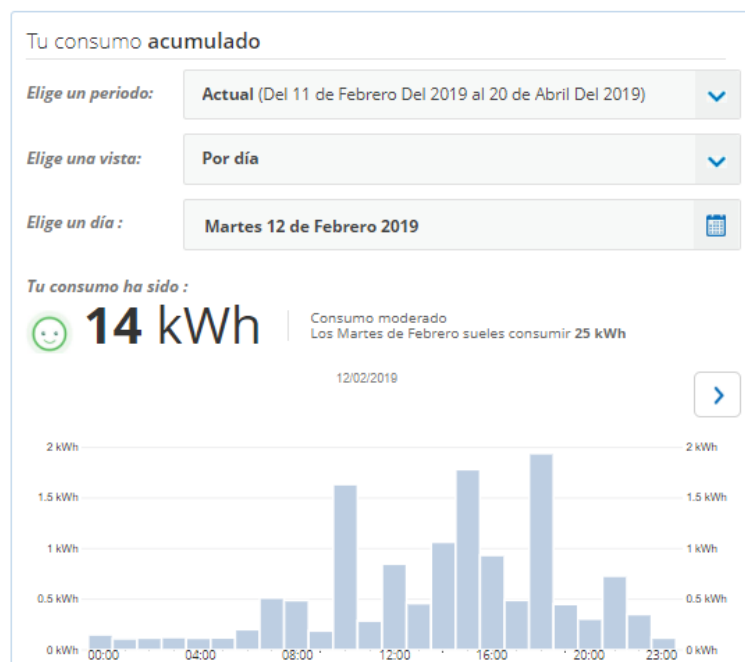


Figura 4.1: Panel de control del área cliente Endesa

- **Consumo interno del sistema ( $C_{int}$ )**

El sistema propuesto tiene un consumo constante de funcionamiento, cuyo valor se ha estimado en aproximadamente 2 Kw al día, alrededor de unos 0,088 vatios por hora. Consta del consumo por funcionamiento de placas fotovoltaicas y realización de carga y descarga de batería.

- **Carga de batería (CB)**

Cantidad de energía que se almacena en la batería para su posterior uso (almacenaje en batería). Esta variable cobra sentido en el caso de un abaratamiento de alguna fuente de generación de energía, lo que propicia que se obtenga más de la necesitada y se almacene para cuando el precio sea mayor.

- **Vertido al mercado eléctrico (CR)**

Cantidad de energía que se vende al mercado eléctrico. Como particular, se puede disponer de una instalación fotovoltaica y verter energía a la red eléctrica, aunque es una práctica sujeta a numerosas trabas legales y dificultades en las que no se entrará en el desarrollo de este trabajo fin de grado. Esta energía se vertería al intramercado de red conocido como el mercado SPOT, aquel donde los activos que se compran o venden se entregan al precio de mercado del momento de la compra/venta.

Como se puede observar, el vertido al mercado eléctrico es un consumo que tiene un beneficio económico que ha de tenerse en cuenta. Existe una retribución por Kw vertido a la red dependiente del momento del día, ya que como se ha comentado antes, el valor de compra/venta del mercado SPOT varía. Para la obtención de estos valores se vuelve a hacer uso de la ya mencionada API e-sios, proporcionando a la función *get\_incoming\_prices* el indicador del precio SPOT, presente en el fichero de constantes del proyecto. Análogo a la obtención del PVPC, se retorna un buffer con los 24 valores requeridos del precio SPOT correspondientes a las 24 horas a simular.

Aunque a priori parezca que el hecho de cargar las baterías no tiene una compensación económica, esto no es del todo correcto. Existe un beneficio económico, aunque no directo, con esta práctica. Puede ser explicado como la cantidad ahorrada por almacenar esa energía y no consumirla, ya que se ha pagado por ella. Este valor puede verse como el mínimo de los precios de las fuentes de generación de energía en el momento de la carga. Veamos un breve ejemplo:

En la hora  $t$  se ha obtenido energía fotovoltaica a un precio de 0,11 € el Kwh. Por otro lado, se ha obtenido

energía de la compañía eléctrica contratada a un precio de 0,14 € el Kwh. El beneficio económico indirecto por cargar un Kw de energía en batería en esta hora  $t$  será de 0,11 €.

#### 4.1.3. Variables de control

Los valores de las variables definidas anteriormente son los que se intentan optimizar, pero existe otro conjunto de variables conocido como las variables de control. Aunque se denotan como variables, en el caso concreto de una simulación son constantes, ya que sus valores están predefinidos para la simulación del modelo. Este conjunto está formado por los valores de los que dependen las variables de entrada y salida y en función de los que se busca una optimización, y caracterizan tanto la simulación como la situación en una hora determinada.

Este conjunto está formado por:

- **Fecha de inicio**

Valor que hace referencia al inicio de la simulación. Este valor es representado mediante el módulo `datetime` de Python. `Datetime` [8] es un módulo de la librería estándar de Python que permite manipular y trabajar con fechas. Este valor será un día y una hora de ese día.

- **Fecha de fin**

Corresponde al fin de la simulación. Siempre será 24 horas a partir de la fecha de inicio. Al igual que el anterior, se representa haciendo uso de `datetime`.

- **Número de módulos fotovoltaicos**

El número de módulos fotovoltaicos juega un papel fundamental. A mayor número de módulos, se producirá mas energía, pero mayor deberá ser la inversión para adquirirlos.

- **Precio de un módulo fotovoltaico**

En este trabajo el tipo de módulo fotovoltaico sera el se suele usar a nivel particular, de tamaño pequeño, con una potencia nominal en condiciones ideales de 50 watios hora 4.2. Este modulo tiene un precio por unidad de 40 €, por lo tanto este valor tendrá el mismo valor en todas las simulaciones.



Figura 4.2: Panel fotovoltaico de 50 W

- **Años en amortizar la inversión de los módulos fotovoltaicos**

El número de años en los que se desea amortizar la inversión realizada en la adquisición de los módulos fotovoltaicos, exclusivamente mediante su uso. Como se ha comentado anteriormente, no es algo trivial ya que determinará en gran medida el precio de extracción de energía fotovoltaica.

- **Precio de la batería**

En este trabajo el tipo de batería usado será una batería estacionaria 4.3 compuesta por plomo abierto

y gel. Este tipo de batería esta compuesta por dos vasos de 2V cada uno que disponen de un amplio rango de autonomía y una vida útil bastante larga, alrededor de unos 20 años. Son aconsejadas en instalaciones con un consumo medio (microondas, horno, lavadora, aire acondicionado, etc), es decir, perfectas para un hogar de tamaño normal. Cómo su tensión es de 2V, se debe instalar un total de 6 vasos en serie, al estar la instalación solar a 12V. Su precio es elevado debido a la gran capacidad, siendo un precio de 3900 € el de la obtención de los 6 vasos.



Figura 4.3: Batería estacionaria

- **Capacidad de la batería**

El tipo de batería usado, es decir, batería estacionaria de 6 vasos, tiene una capacidad total de aproximadamente 21 Kw. La profundidad de descarga de este tipo de batería es aproximadamente del 50 %, esto es, como se comento durante la explicación de las variables de entrada y salida, el tanto por ciento que se puede descargar dicha batería sin resultar perjudicial para su salud y por lo tanto afectar a su ciclo de vida útil.

- **Nivel de carga inicial de la batería**

Variable de control que define el estado inicial de la batería a la hora de realizar la simulación del modelo. Esto es la cantidad de energía que tiene almacenada la misma.

- **Años en amortizar la inversión de la batería**

Como ocurre en el caso de la inversión fotovoltaica, se debe determinar el número de años en los que se desea realizar la amortización de la inversión por adquirir la batería. Al tratarse de un valor mucho mas elevado debe ser mayor al del caso anterior, ya que si no se dispararía el precio de descargar las baterías y dejaría de ser una entrada a tener en cuenta al no resultar rentable.

Con esto quedan identificadas cada una de las variables que entran en juego en el modelo, así como su medio de adquisición.

## 4.2. APLICACIÓN DE LÓGICA DIFUSA PARA LA DETERMINACIÓN DE LOS ESTADOS METEOROLÓGICOS

En la segunda iteración se describe el procedimiento llevado a cabo para el uso de los estados meteorológicos obtenidos de la API de AEMET [9].

Tal y como se comentó en la iteración anterior, la respuesta procesada en cada caso a la petición *requests* consta de estados meteorológicos en formato de cadena de texto. Un ejemplo de respuesta del tipo texto en lenguaje natural se vió en el listado 4.3. En el listado 4.5 se muestra (simplificada) la respuesta recibido en el caso de la consulta al día en curso. (©AEMET)

**Listado 4.5:** Ejemplo de respuesta de la API - AEMET para el día en curso

```
{
  origen: {
    productor: "Agencia Estatal de Meteorología - AEMET. Gobierno de ↵
      ↵ España",
    web: "http://www.aemet.es",
    language: "es",
    copyright: "AEMET. Autorizado el uso de la información y su ↵
      ↵ reproducción citando a AEMET como autora de la misma.",
    notaLegal: "http://www.aemet.es/es/nota_legal"
  },
  elaborado: "2019-2-12",
  nombre: "Consuegra",
  provincia: "Toledo",
  prediccion: {
    dia: [
      {
        estadoCielo: [
          {
            periodo: "08",
            descripcion: "Cubierto"
          },
          {
            periodo: "09",
            descripcion: "Cubierto con lluvia escasa"
          },
          {
            periodo: "10",
            descripcion: "Cubierto con lluvia escasa"
          },
          {
            periodo: "11",
            descripcion: "Cubierto"
          },
          ...
        ]
      }
    ]
  }
}
```

Cómo se puede observar se diferencia de la respuesta en formato texto mostrada en el listado 4.3, pues se trata de una respuesta en formato json. El lenguaje json es un formato de texto simple que se utiliza para el intercambio de información y es tomado como lenguaje independiente, aunque tuvo sus inicios en Javascript, haciendo honor a su nombre (*Javascript Object Notation*). En el campo 'predicción' existe un subcampo 'estadoCielo' (entre otros que se han obviado por no ser de interés en este trabajo) que contiene una lista con los estados meteorológicos de la previsión. Cada elemento de la lista contiene dos valores: período (hora del día de esa previsión) y descripción (cadena de texto que describe el estado del cielo, del mismo conjunto de palabras que se emplea para encontrar las ocurrencias en el texto en el otro tipo de respuesta). Claramente existe un problema con los buffers de estados meteorológicos que se obtienen de procesar las peticiones a la API AEMET, ya que para la determinación de la máxima energía fotovoltaica posible en una hora  $t$  debe conocerse la potencia nominal posible (potencia que es capaz de suministrar el módulo fotovoltaico), directamente proporcional al estado meteorológico, el cual es un texto que describe la situación y no un valor numérico que representa los vatios que puede dar un módulo en esas condiciones, y a priori no se dispone de una forma directa de relacionarlas.



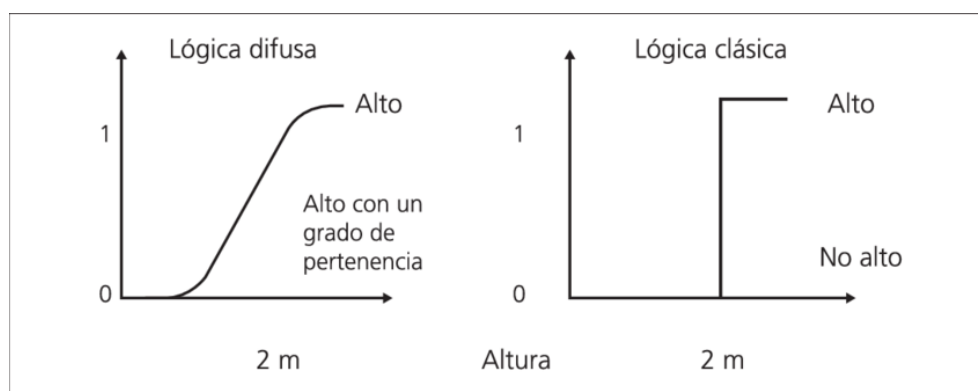
Se emplea **lógica difusa** para poder resolver la problemática mencionada anteriormente.

#### 4.2.1. Lógica difusa

La teoría de la lógica difusa proporciona un marco matemático que permite modelar la incertidumbre de los procesos cognitivos humanos para poder ser tratable por un computador. Estos procesos cognitivos hacen referencia a expresiones del tipo:

- Si no vives *lejos* puedes ir en bicicleta.
- Si hace *mucho* frío llévate un chaquetón.

Los humanos son capaces de interpretar estos valores rápidamente. Sin embargo, las máquinas tienen algún que otro problema, debido a que no existe un valor cuantitativo que indique la distancia a la que se refiere la palabra *lejos* o cuánto es *mucho* frío. Si se intentan trasladar estas reglas a código, aparecen dificultades ya que no se puede procesar numéricamente. Una opción es definir intervalos de valores que comprende cada palabra (por ejemplo, tomando *lejos* como la distancia comprendida entre 5 y 10 kilómetros), pero esto no es preciso ya que para un computador, la distancia de 5,01 kilómetros sería *lejos* rotundamente, cuando en realidad la interpretación correcta no es así. Con esto queda a la vista que la lógica convencional no trata de forma eficiente este problema presentando numerosas limitaciones. Otro ejemplo típico es el de la imagen 4.4, donde se puede observar cómo la lógica clásica interpretaría erróneamente el hecho: *Una persona de dos metros es alta*, pues clasificaría una persona de 1,99 metros como no alta, mientras que la lógica difusa lo clasificaría mediante un grado de pertenencia. La solución pasa por emplear un método de razonamiento afín a la lógica difusa.



**Figura 4.4:** Lógica clásica vs lógica difusa

La lógica difusa [19] permite representar matemáticamente la **incertidumbre**. Según Zadeh [26], "Cuando aumenta la complejidad, los enunciados precisos pierden su significado y los enunciados útiles pierden precisión.", es decir, *los árboles no te dejan ver el bosque*, pues prácticamente cualquier problema del mundo puede resolverse partiendo de unas variables de entrada y buscando obtener como objetivo un conjunto de variables de salida. La lógica difusa establece esta relación entre variables de forma correcta.

#### 4.2.2. Fuzzy Sets

En la teoría de conjuntos de la lógica clásica, el grado de pertenencia puede tomar solo los valores 0 y 1, que representan que el elemento pertenece o no pertenece al conjunto. En la lógica difusa existe el concepto de **conjunto difuso** [25], establecido por Zadeh. Para trabajar con valores difusos se realiza un proceso

denominado *fuzzificación* que da resultados difusos. Estos resultados se someten a un proceso de *defuzzificación* para transformarse en valores discretos (llamados *crisp*), que tendrán un **grado de pertenencia** a los conjuntos difusos el cuál será un valor en el intervalo  $[0, 1]$ , y representa cuánto pertenece al conjunto.

Así pues, hay un claro ejemplo de conjuntos difusos con los estados meteorológicos. En la Figura 4.5 la API de AEMET [9] define los siguientes estados meteorológicos posibles (imagen obtenida de ©AEMET). Como se puede observar existe un gran número de estados meteorológicos posibles, pero no todos deben ser conjuntos difusos. Esto es porque en un gran número de ellos un módulo fotovoltaico no genera energía, debido a que son estados con tiempo muy contraproducente para la energía solar fotovoltaica. En estos estados el valor máximo a generar por los módulos será de 0 vatios. Los estados favorables (donde un módulo genera una potencia mayor a 0 vatios) serán tomados como conjuntos difusos. En la tabla 4.1 se muestran cuáles son estos estados.

Por ejemplo, no se puede determinar cuántos vatios se producen como máximo con un tiempo *Despejado* o *Cubierto con nubes altas*, pero podemos mostrar los conjuntos difusos y gráficamente comprobar su distribución, para obtener un valor discreto de cada conjunto difuso, denominado el **centroide** o centro de gravedad del conjunto difuso. Este proceso es conocido como razonamiento aproximado a partir de inferencia difusa. La inferencia difusa permite obtener un valor de salida para un valor de entrada empleando la teoría de conjuntos difusos.

















































Un ejemplo de cuestión a resolver es: *¿Qué potencia pico se puede obtener de un módulo fotovoltaico si hay intervalos nubosos?*. Como hemos comentado anteriormente, la respuesta sería el centroide del conjunto difuso *Intervalos Nubosos*.

El centroide es el punto que divide el conjunto difuso en dos partes de igual masa. En la ecuación 4.4 se muestra el procedimiento para calcularlo, realizando el sumatorio de las potencias tomadas por su grado de pertenencia al conjunto dividido entre el sumatorio de dichos grados de pertenencia. En la tabla 4.1 se incluye el valor del centroide de cada conjunto difuso.

$$\text{Centroide} = \frac{\sum_{x=i}^n x \mu_A(x)}{\sum_{x=i}^n \mu_A(x)} \quad (4.4)$$

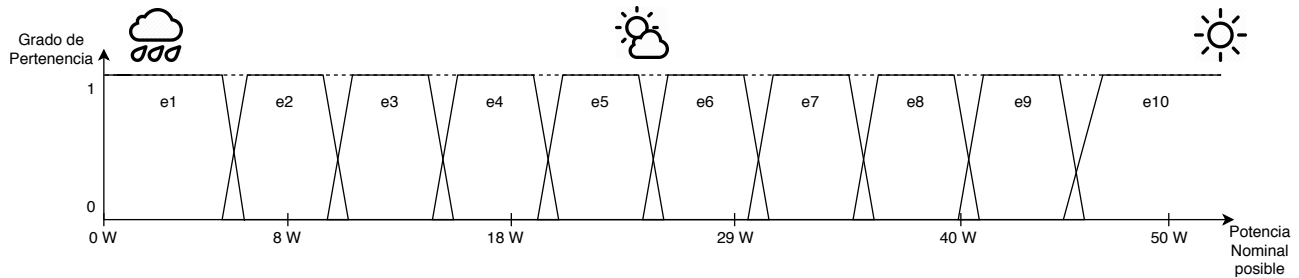
Para representar los conjuntos difusos, se parte del hipotético caso en el que cada conjunto difuso desciende de manera ligeramente mas inclinada que asciende, representando el factor de adaptación de un módulo fotovoltaico a un nuevo estado meteorológico. En la figura 4.6 se puede observar el gráfico de los conjuntos difusos definidos en la tabla 4.1. En el eje horizontal se encuentra la potencia máxima posible del módulo fotovoltaico en vatios y en el eje vertical el grado de pertenencia a los conjuntos difusos  $[0,1]$ . Si avanzamos en el eje horizontal, vemos que el estado meteorológico es cambiante hacia estados favorables, y viceversa, ya que esto es directamente proporcional a la máxima potencia posible del módulo.

## Estado del cielo

			
Despejado	Despejado noche	Poco nuboso	Poco nuboso noche
			
Intervalos nubosos	Intervalos nubosos noche	Nuboso	Nuboso noche
			
Muy nuboso	Cubierto	Nubes altas	Nubes altas noche
			
Intervalos nubosos con lluvia escasa	Intervalos nubosos con lluvia escasa noche	Nuboso con lluvia escasa	Nuboso con lluvia escasa noche
			
Muy nuboso con lluvia escasa	Cubierto con lluvia escasa	Intervalos nubosos con lluvia	Intervalos nubosos con lluvia noche
			
Nuboso con lluvia	Nuboso con lluvia noche	Muy nuboso con lluvia	Cubierto con lluvia
			
Intervalos nubosos con nieve escasa	Intervalos nubosos con nieve escasa noche	Nuboso con nieve escasa	Nuboso con nieve escasa noche
			
Muy nuboso con nieve escasa	Cubierto con nieve escasa	Intervalos nubosos con nieve	Intervalos nubosos con nieve noche
			
Nuboso con nieve	Nuboso con nieve noche	Muy nuboso con nieve	Cubierto con nieve
			
Intervalos nubosos con tormenta	Intervalos nubosos con tormenta noche	Nuboso con tormenta	Nuboso con tormenta noche
			
Muy nuboso con tormenta	Cubierto con tormenta	Intervalos nubosos con tormenta y lluvia escasa	Intervalos nubosos con tormenta y lluvia escasa noche
			
Nuboso con tormenta y lluvia escasa	Nuboso con tormenta y lluvia escasa noche	Muy nuboso con tormenta y lluvia escasa	Cubierto con tormenta y lluvia escasa

**Figura 4.5:** Estados del cielo posibles AEMET

Cada conjunto difuso es una función PI o **trapezoidal**, ya que no existe un único punto donde el grado de pertenencia al conjunto es 1, si no que se mantiene ese valor de pertenencia hasta que el módulo comienza a experimentar un cambio en el estado del cielo y se debe adaptar a dicho nuevo estado.



**Figura 4.6:** Fuzzy sets de los estados meteorológicos

Tal y cómo se comentó en la iteración anterior, se utilizan módulos fotovoltaicos con una potencia nominal máxima de 50 vatios, que solo podría ser alcanzada en el estado meteorológico óptimo (*Despejado*), y en función de este dato, se toman los valores para obtener los centroides.

Variable	Conjunto difuso	Centroide
e10	Despejado	48 W
e9	Poco nuboso	43.16 W
e8	Nubes altas	38.16 W
e7	Intervalos nubosos	33.16 W
e6	Intervalos nubosos con lluvia escasa	28.16 W
e5	Intervalos nubosos con lluvia	23.16 W
e4	Nuboso	18.16 W
e3	Nuboso con lluvia escasa	13.16 W
e2	Cubierto	8.16 W
e1	Cubierto con lluvia escasa	2.66 W

**Tabla 4.1:** Estados meteorológicos favorables

Estos valores se almacenarán en un diccionario, disponible en el fichero de constantes del proyecto (módulo *project\_constants*). Dicho diccionario será usado para realizar el parseo de los estados meteorológicos obtenidos de la API AEMET (cadenas de texto) a valores cuantitativos (centroide del conjunto difuso), y poder ser usables por el sistema para determinar la **máxima energía fotovoltaica** que se puede obtener en un momento determinado.

### 4.3. CREACIÓN DE LAS RELACIONES Y RESTRICCIONES PROPIAS DEL MODELO

Cómo se comento en el capítulo relativo al Objetivo del trabajo fin de grado, éste se aborda como un **problema de satisfacción de restricciones**.

La programación por restricciones es una metodología software que permite resolver problemas de gran complejidad, típicamente NP. Ésta metodología ha generado mucha expectación en el área de la inteligencia artificial desde la década de los 60, ya que tiene un gran potencial para la resolución de problemas reales. La idea básica de la programación por restricciones es primero declarar una serie de restricciones sobre el dominio del problema que atañe, para después dar con soluciones que satisfacen las anteriores restricciones de la forma más optima posible. Así, un problema de satisfacción de restricciones [24] está caracterizado por:

- Un conjunto de variables, donde cada variable dispone de un dominio de valores que puede tomar.
- Un conjunto de restricciones, que permite conocer las posibles combinaciones de las variables.
- La solución al PSR será la asignación de valores a las variables de forma que se satisfacen las restricciones y se alcanza el objetivo, representado típicamente como una función a optimizar.

Las restricciones se caracterizan por su **aridad**, que viene a ser el número de variables que involucra. Pudiendo ser unarias, si solo involucran una variable; binarias, si involucran dos variables; y n-arias, si involucran más de dos variables. Se deben tener en cuenta un tipo de restricción adicional, ya que están presentes en este trabajo, como son las **restricciones lineales**, expresadas teóricamente de la forma 4.5

$$\sum_i^n a_i x_i (<, \leq, =, \geq, >, \neq) c \quad (4.5)$$

siendo  $a$  el coeficiente de la variable  $x$  y  $c$  constante.

#### 4.3.1. Variables del PSR

En las iteraciones 1 y 2 se identificaron las variables del problema, las cuáles están divididas en tres grupos: variables de entrada, variables de salida y variables de control. Sin embargo conviene mencionar cuáles de estas variables son las propias del problema de satisfacción de restricciones, y cuáles formarán las restricciones. El objetivo del problema es obtener valores de energía para los paneles fotovoltaicos, baterías y red eléctrica, de tal modo que se cubra la demanda energética del hogar y que el gasto económico sea el menor. Es por esto que las variables propias del problema de satisfacción de restricciones serán:

- **Energía fotovoltaica (EF).** Energía obtenida de los módulos fotovoltaicos. 4.6

$$Dom_{EF} = [0, EF_{max}] \quad (4.6)$$

- **Energía de red (ER).** Energía importada de la red eléctrica. 4.7

$$Dom_{ER} = [0, +\infty) \quad (4.7)$$

- **Energía de batería (EB).** Energía obtenida de la batería. 4.8

$$Dom_{EB} = [0, +\infty) \quad (4.8)$$

- **Consumo de batería (CB).** Energía consumida para cargar la batería. 4.9

$$Dom_{CB} = [0, +\infty) \quad (4.9)$$

- **Consumo de red (CR).** Energía vertida a red a cambio de retribución económica. 4.10

$$Dom_{CR} = [0, +\infty) \quad (4.10)$$

Las restricciones estarán definidas en función de dichas variables y cada solución al problema estará formada por un valor para cada una de estas variables. Éstos valores satisfacen las restricciones y además serán los óptimos para que se produzca el menor gasto económico posible. El resto de variables (variables de control) determinarán las propias restricciones y el valor de las anteriores dependerá de éstas en una hora concreta  $t$ , entre 0 y 24h.

### 4.3.2. Restricciones del PSR

A continuación se determinan las restricciones a las que está sometido el modelo en una hora  $t$ :

- **Toda la energía generada debe ser consumida. 4.11**

Hace referencia al principio básico de la energía, la energía que se produce se consume de un modo u otro, no es posible que la suma de las variables correspondientes a la generación de energía (EB, ER y EF) sea distinta a la suma de las variables que hacen referencia al consumo de energía (CR, CB,  $C_{int}$  y C). Ésto debe producirse en cada una de las horas de la simulación. Así, tenemos una restricción lineal y n-aria correspondiente a la suma de las 24 horas correspondientes a una simulación, por lo que ésta restricción a efectos prácticos es tomada como 24 restricciones a cumplir.

$$\sum_{i=0}^{23} EF_i + ER_i + EB_i = CR_i + CB_i + C_{int} + C \quad (4.11)$$

- **No se puede producir energía fotovoltaica durante la noche. 4.12**

Algo obvio, pues sin luz solar la energía fotovoltaica no es posible. Ésto no está controlado en la API AEMET, ya que las peticiones relativas a la noche no reflejan una descripción propia del tiempo nocturno, si no que devuelve los mismos valores independientemente de si existe luz solar, por lo que debe manejarse mediante una restricción. Para este trabajo las horas de la noche serán las pertenecientes al intervalo temporal desde las 22:00 pm hasta las 7:00 am. Cómo posible trabajo futuro, podría determinarse este intervalo en función de la estación del año para que pueda ser un intervalo con mayor grado de efectividad. Se trata de una restricción unaria, donde para ciertos valores de  $t$ , EF debe ser 0. Es por esto que esta restricción a efectos prácticos es tomada como 9 restricciones (las 9 horas de noche definidas anteriormente)

$$EF_{noche} = 0 \quad (4.12)$$

- **La energía fotovoltaica generada no puede ser mayor que la máxima energía fotovoltaica en  $t$ . 4.13**

No se puede superar el umbral de generación de energía fotovoltaica establecido por la potencia nominal máxima de esa hora  $t$ , pues se estaría violando la capacidad real de producción de los módulos fotovoltaicos del sistema. Es una restricción lineal unaria, ya que la energía fotovoltaica máxima de cada hora  $t$  es constante, pues como se comentó anteriormente, sólo es dependiente del número de módulos fotovoltaicos y la situación meteorológica (obtenida de la API AEMET). A efectos prácticos, esta restricción es tomada como 24 restricciones a cumplir referentes a las 24 horas de la simulación.

$$\sum_{i=0}^{23} EF_i \leq EF_i^{max} \quad (4.13)$$

- **La energía obtenida de la batería no puede ser mayor que el nivel de batería actual teniendo en cuenta la profundidad máxima de descarga. 4.14**

Básicamente no se puede obtener una cantidad de energía mayor a la posible en esa hora  $t$ , que vendrá determinada por la diferencia entre el nivel de carga disponible al comienzo de esa hora y la capacidad máxima de la batería por la profundidad de descarga (50 %), para evitar daños en su ciclo de vida útil. Restricción unaria, pues solo involucra la variable EB, ya que el resto de elementos de la restricción son constantes en una hora  $t$  (nivel de carga actual, capacidad máxima de la batería

y profunda de descarga). Al igual que las restricciones anteriores es lineal y a efectos prácticos representa 24 restricciones a cumplir.

$$\sum_{i=0}^{23} EB_i \leq nivel_{i-1} - capacidad_{max} * profundidad_{descarga} \quad (4.14)$$

- **El consumo para cargar la batería no puede ser mayor que la capacidad de la misma menos el nivel restante después de t. 4.15**

Parecido a la restricción anterior, en esta se modela el hecho de cargar la batería (CB) en cada hora t, el cuál está condicionado por la cantidad de batería restante para completar la carga (100 %), obtenido mediante la diferencia entre la capacidad máxima de la misma y lo consumido en la hora t (nivel de carga antes de comenzar la hora t menos la energía consumida de batería en t). Restricción binaria pues involucra tanto el consumo de batería (CB) como la energía de batería (EB), siendo la capacidad máxima de la batería y el nivel de carga en t constantes. Es tomado como 24 restricciones ya que debe cumplirse en cada una de las 24 horas de una simulación.

$$\sum_{i=0}^{23} CB_i \leq capacidad_{max} - (nivel_{i-1} - EB_i) \quad (4.15)$$

Por lo tanto, a efectos prácticos, el PSR tiene 81 restricciones que satisfacer para determinar los valores de las variables.

### 4.3.3. Función Objetivo

Cómo se ha comentado antes, un problema de satisfacción de restricciones está determinado por un conjunto de variables y sus dominios, un conjunto de restricciones de esas variables, y un objetivo. Por el momento se dispone de los dos primeros, por que en este apartado se procede a determinar el último, **la función objetivo**.

Un PSR que cuenta únicamente con variables y restricciones para esas variables podrá tener numerosas soluciones, representadas como una tupla con valores para cada variable. Añadiendo un objetivo al PSR se consigue unificar la solución, pues de todas esas soluciones, sólo una optimizará un objetivo concreto, y contendrá los valores óptimos de cada variable para ello.

Como se ha comentado a lo largo del desarrollo de este trabajo fin de grado, el objetivo del problema es **minimizar el gasto económico** producido mediante la optimización de energía en cada simulación de 24 horas, por lo tanto se buscarán valores de las variables que, además de satisfacer el conjunto de restricciones, sean óptimos para que el gasto económico sea mínimo. Éste gasto económico es dependiente del precio en la hora t de cada una de las energías que representan las variables. De estos precios se habló y se implementó la forma de obtenerlos en la iteración 1. Sus valores son constantes en cada hora t. Dicho esto, la función objetivo a minimizar está formada por el sumatorio de los gastos económico en cada hora t, por lo que representa el gasto económico de toda la simulación. 4.16

$$f(x) = \sum_{i=0}^{23} EF_i P_i^F + ER_i P_i^{RPVPC} + EB_i P_i^{Bout} - CB_i P_i^{Bin} - CR_i P_i^{RSPOT} \quad (4.16)$$

Siendo:

- $EF_i P_i^F$ : Gasto económico producido al generar energía fotovoltaica en la hora t.
- $ER_i P_i^{RPVPC}$ : Gasto económico producido al importar energía a la compañía eléctrica en la hora t.



- $EB_i P_i^{B_{out}}$ : Gasto económico producido al descargar la batería en la hora  $t$ .
- $CB_i P_i^{B_{in}}$ : Ganancia económica producida al cargar la batería en la hora  $t$ .
- $CR_i P_i^{R_{SPOT}}$ : Ganancia económica producida al verter energía al mercado eléctrico en la hora  $t$ .

En cada simulación se buscará que el valor de  $f(x)$  sea el menor posible y la solución al PSR de esa simulación será los valores que deben tomar las variables para hacerlo posible.

#### 4.3.4. Implementación de la clase Simulation

Vistos los apartados anteriores, es la hora de implementar una clase para representar el modelo de simulación, y poder crear objetos que representan una simulación concreta. Esta clase se llama **Simulation** 4.7 y está disponible en el módulo simulation.

Los atributos de clase de Simulation serán las variables de control que tendrá cada objeto correspondiente a una simulación:

- start: variable de control referente a fecha de inicio de la simulación. Es de tipo datetime.
- end: variable de control referente a fecha de fin de la simulación. Al igual que start, es una variable de formato fecha (datetime).
- ef\_price: variable de control referente al precio de generar energía fotovoltaica. Se trata de un número tipo float, pues tendrá el mismo valor en toda la simulación (se obtiene a partir de la inversión realizada)
- er\_price: variable de control referente a los precios que tendrá el PVPC en cada hora de la simulación. Es una lista de 24 elementos de tipo float.
- eb\_price: variable de control referente al precio de descargar la batería. Es un número de tipo float ya que será igual en las 24 horas de la simulación.
- cr\_price: variable de control referente a los precios SPOT en cada hora de la simulación, es decir, el precio del vertido de energía a la red eléctrica.
- cb\_price: variable de control referente a los precios de cargar la batería. Como es dependiente de los precios de energía fotovoltaica y de red de cada hora  $t$ , se trata de una lista con 24 valores.
- battery\_capacity: variable de control referente a la capacidad total de la batería. Variable de tipo float.
- battery\_level: variable de control que representa el nivel inicial de carga de la batería. Variable de tipo float.
- discharge\_depth: variable de control referente a la profundidad de descarga permitida en la batería. Variable de tipo float.
- max\_ef\_buffer: Más que una variable de control, representa los 24 valores máximos posible de energía fotovoltaica, obtenidos como se comentó anteriormente mediante la información de la API AEMET y el número de módulos fotovoltaicos, por ende, se trata de una lista de 24 valores.
- c\_int: referente a la variable de control del consumo interno del sistema. Variable de tipo float.



- c: referente al consumo del hogar, lista de los 24 valores con el consumo del hogar en las 24 horas de la simulación.

Las funciones de esta clase sirven para calcular algunos de los atributos anteriores. En la figura 4.7 se muestra la clase UML de Simulation.

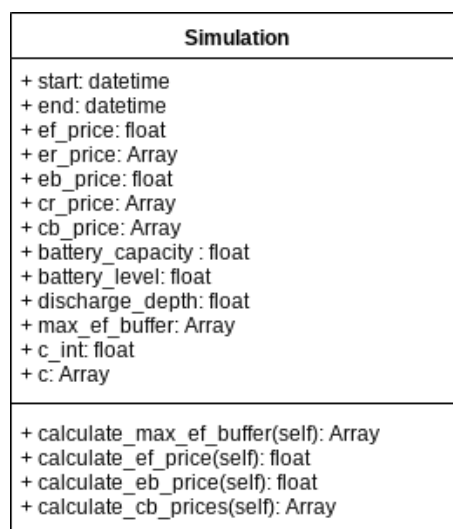


Figura 4.7: Clase Simulation

En la próxima iteración se implementará cada una de las restricciones para permitir ejecutar la simulación.

## 4.4. GENERACIÓN OPTIMIZADA DE ENERGÍA MEDIANTE PROGRAMACIÓN LINEAL

Cómo se comentó anteriormente, éste PSR dispone de 81 restricciones. Además, no interesa cualquiera de las soluciones posibles si no que debe buscarse la solución más óptima de todas. Por lo tanto para su resolución deberá emplearse algún procedimiento lo suficientemente potente para contemplar ambos requisitos.

### 4.4.1. Programación Lineal

La programación lineal [22] tiene como objetivo optimizar una función lineal cuyas variables están sujetas a un conjunto de restricciones lineales. Se trata de un campo de la matemática muy efectivo para la resolución de problemas donde se desea sacar el mayor provecho de una situación.

Históricamente, el concepto de programación lineal debe su nombre a John Von Neumann (1947), uno de los matemáticos más importantes del siglo XX gracias a sus contribuciones en las ciencias de la computación; y a George Dantzig (1947), cuyo trabajo intentaba asignar 70 puestos de trabajo a 70 personas mediante programación lineal. Las permutaciones necesarias para la asignación óptima de dichos puestos era factorial de 70 (70!), algo enorme, pues el número de combinaciones de variables es inmenso. Curiosamente, mediante programación lineal el problema se resuelve en un momento pues el número de combinaciones se reduce en su mayor parte. La programación lineal puede ser aplicable a numerosos problemas comunes tales como:

- Asignación de horarios a profesores en un centro educativo para obtener la mayor productividad a la par que comodidad para profesor y alumno.
- Distribución de elementos en almacenes de tal modo que se reduzca el costo de almacenamiento teniendo en cuenta la limitada capacidad.
- Distribución de bienes entre compradores y consumidores de tal modo que las ganancias del intermediario sean máximas.

Cómo se puede observar, el problema de este trabajo fin de grado está muy relacionado con el último ejemplo, pues se distribuye cantidad de energía entre fuentes de entrada y fuentes de salida de manera óptima para garantizar un gasto mínimo de consumo energético.

Para un problema de programación lineal pueden existir varios casos en su resolución:

- Existe una solución óptima.
- Existen varias soluciones óptimas.
- No existe solución.
- Existen soluciones infinitas.

La situación deseada es la primera, pero puede ocurrir alguno de los otros casos. Estas situaciones pueden resolverse convirtiendo las restricciones que son inecuaciones (desigualdades) en igualdades.

Existen varios métodos de programación lineal. El más utilizado es conocido como el **método Simplex**, pues es muy potente debido a que se basa en evaluar solo algunos puntos extremos mediante dos condiciones:

- **Optimalidad.** La solución inferior relativa al punto de solución actual no se tiene en cuenta.
- **Factibilidad.** Una vez se encuentra una solución básica factible, sólo aparecerán soluciones factibles.

Otro método de programación lineal es el método de ramificación y acotamiento *branch and bound*, el cual divide el problema en varios subproblemas de programación lineal, acotamiento que permite obtener soluciones óptimas que se mejoran por cada subproblema.

En el ámbito de las ciencias de la computación existen librerías que permiten emplear algoritmos de programación lineal para la resolución de problemas de optimización. En este trabajo fin de grado se empleará **SciPy**, un ecosistema de librerías de código abierto con numerosas herramientas para matemáticas, ciencia e ingeniería.

#### 4.4.2. Optimización con SciPy

Scipy [16] proporciona un conjunto de paquetes de computación científica para el lenguaje Python como son Numpy, Scipy, Matplotlib, iPython, SymPy y Pandas. En este caso el trabajo se centra en el módulo Scipy.optimize [15], que contiene las herramientas de Scipy para optimización. Proporciona numerosos algoritmos de optimización para uso común:

- Minimización sin restricciones y restringida de funciones escalares multivariadas.
- Optimización global mediante fuerza bruta.

- Minimización de mínimos cuadrados.
- Minimización de funciones univariantes escalares y búsqueda de soluciones.
- Solución de sistemas de ecuaciones multivariantes con una gran cantidad de algoritmos.

Para el caso propio de este trabajo fin de grado en que se el objetivo es minimizar una función sujeta a un gran conjunto de restricciones, lo más conveniente es hacer uso del módulo **linprog** de `Scipy.optimize`, específico para trabajar con programación lineal. Resuelve problemas del tipo definido en el listado 4.6, donde:

- `A_ub` representa los coeficientes de las restricciones definidas como inecuaciones.
- `b_ub` representa las constantes del tipo de restricción inecuación.
- `A_eq` representa los coeficientes de las restricciones de igualdad, es decir, ecuaciones.
- `b_eq` representa las constantes del tipo de restricción de igualdad.
- `(lb, ub)` representan los límites inferior y superior del dominio de la variable `x`.
- `c` es la función a minimizar, dependiente de la variable `x`.

El caso particular de este trabajo se adapta perfectamente a dicho modelo de problema. Pero antes de

**Listado 4.6:** Tipo de problema aplicable a `Scipy.optimize.linprog`

```
# Minimizar:
c @ x
# Sujeto a:
A_ub @ x <= b_ub
A_eq @ x == b_eq
lb <= x <= ub
```

implementar el algoritmo `linprog`, se deben implementar cada una de las restricciones del modelo, algo complejo en este caso pues existen numerosas restricciones al tratarse de una función lineal, pues cada una de las variables involucradas en la función a minimizar 4.16 tendrá realmente 24 valores, correspondientes a las 24 horas de la simulación, y desde el punto de vista de la implementación, será tenido en cuenta como 24 variables distintas.

Antes de implementar cada una de las restricciones, se debe hacer una modificación en la clase `Simulation`. Se añaden cinco nuevos atributos a la clase:

- **f**: Esta variable representa la función objetivo (véase la ecuación 4.16), expresada como una lista con los coeficientes de cada variable en la función los cuáles representan el precio del tipo de energía asociado a la variable. Al tratarse de un sumatorio de 24 iteraciones y contener 5 variables en la expresión, esta lista contendrá 120 elementos resultantes de la suma entre los 24 valores de cada una de las variables. Para dotar de valores a la lista se ha implementado la función `generate_function_coefficients()`, que mediante 24 iteraciones concatena a la lista el valor correspondiente de cada coeficiente de variable, que se encuentran en los atributos de clase definidos en la iteración 3 (Véase la representación UML de `Simulation` en la figura 4.7). Esta variable se corresponde con `c` en el modelo de problema para `Scipy Linprog` del listado 4.6.

- **A\_ub, b\_ub:** Cómo se comentó anteriormente representan las restricciones del tipo inecuación. En A\_ub se almacenan en una lista los coeficientes de las inecuaciones en una lista por restricción, de tal modo que se tiene una lista de listas (lista de dos dimensiones) del tipo: [[coeficientes restr. 1], [coeficientes restr.2], ...]. En b\_ub se tiene una lista con las constantes de las inecuaciones, por lo que el tamaño de b\_ub y A\_ub debe ser igual para que se realice el *matching* de coeficientes con constantes por inecuación.
- **A\_eq, b\_eq:** Similar a las dos listas anteriores, pero en este caso se trata de las restricciones de igualdad. Las lista tienen el mismo formato.

Éstas variables serán primordiales a la hora de ejecutar el algoritmo linprog pues de sus valores serán dependientes los resultados para cada variable. Definidas las variables que contendrán los valores de las restricciones se pasa a la implementación de dichas restricciones.

#### 4.4.3. Implementación de las restricciones del tipo 1

La restricción 1 se corresponde con que toda la energía generada debe ser consumida (Véase la ecuación 4.11). Se trata de una restricción de igualdad, por lo que deben dotarse de valor A\_eq y b\_eq. Es una restricción lineal por lo que desde el punto de vista de la implementación se traduce en 24 restricciones, una por hora de la simulación. En el listado 4.7 se muestra la función *generate\_restriction\_1()* que realiza dicha tarea.

**Listado 4.7:** Restricciones del tipo 1

```

1 def generate_restriction_1(self):
2     for i in range(0, 24):
3         restr_coef = [0]*5*24
4         restr_coef[i*5] = 1
5         restr_coef[i*5+1] = 1
6         restr_coef[i*5+2] = 1
7         restr_coef[i*5+3] = -1
8         restr_coef[i*5+4] = -1
9         self.A_eq.append(restr_coef)
10        self.b_eq.append(self.c_int + self.c[i])

```

Primero se deben mostrar a la izquierda de la restricción las variables y a la derecha las constantes. En A\_eq se debe concatenar una lista por restricción del sumatorio que contendrá los valores 0 o 1 en función de la condición mostrada en el listado 4.8. Como se puede observar en el listado 4.7, por cada

**Listado 4.8:** Condición para dotar de valor los coeficientes

```

Si la variable de esa posición aparece en la restricción
    restr_coef[posición] = 1
Si no
    restr_coef[posición] = 0

```

iteración de las 24 (correspondientes a las 24 horas de la simulación) primero se crea una lista **restr\_coef** con solo valores 0. Ésta lista dispone de 120 elementos, pues la restricción es realmente el sumatorio de 24 restricciones y existen 5 variables en la expresión (EF, ER, EB, CR y CB). Como resultado se obtendrá en A\_eq 24 listas de 120 elementos cada una, de los cuáles todos toman el valor 0 excepto los relativos a las posiciones de las variables que entran en juego en la restricción de esa iteración. Es primordial que se preserve el orden de ordenamiento de las variables en todas las restricciones. Deben tener el mismo orden

que en la función objetivo y tomar 1 si aparecen o 0 si no (Podrán tomar el valor -1 si van precedidas de una resta en la restricción). En el caso de  $b_{eq}$ , se concatenan 24 valores, uno por iteración, correspondientes al valor constante de la restricción de esa iteración.

Obsérvese cómo se realizaría la primera iteración, correspondiente a la hora 0 de la simulación:

*Deben dotarse con  $1 EF_0$ ,  $ER_0$  y  $EB_0$ , pues su coeficiente en la restricción es +1. Deben dotarse con  $-1 CR_0$  y  $CB_0$ , pues su coeficiente es -1 en la restricción. El resto de elementos de la lista deben ser 0 (Correspondientes al resto de coeficientes de variables para  $i=1,2,3..$ ). Ésta lista se concatena en  $A_{eq}$ . En  $b_{eq}$  se concatena el valor constante de esta restricción, que es  $c_{int} + C_i^{prop}$ . Con esta queda implementada la restricción de tipo 1 correspondiente a la hora 0 de la simulación.*

#### 4.4.4. Implementación de las restricciones del tipo 2

La restricción 2 hace referencia a que no se puede producir energía fotovoltaica durante la noche (Véase la ecuación 4.12). En este trabajo fin de grado se definen estos valores como los comprendidos entre las 9:30 pm y las 7:00 am. Como se observa en el listado 4.9, para generar las restricciones de este tipo, en cada iteración se inicializa la lista de 120 valores con ceros de manera análoga a las restricciones de tipo 1. Después, para determinar la hora real correspondiente de la iteración en curso, se debe sumar a la hora de inicio de la simulación el número de iteración actual. El uso del módulo de la librería estándar de Python *datetime* [8] hace que sea posible manejar variables en formato hora o fecha. En las iteraciones en las que la hora actual esté comprendida en las definidas como horas nocturnas, el valor de la posición de EF (energía fotovoltaica) en esa iteración tomará el valor 1. Éstas listas resultantes de cada iteración se van concatenando con  $A_{eq}$ , pues son restricciones de igualdad. En cuanto a  $b_{eq}$ , por cada iteración se concatena un 0, pues el valor constante de esta restricción es 0 debido a que la energía fotovoltaica de noche es nula. Tras la ejecución de la función *generate\_restriction\_2()*  $A_{eq}$  cuenta con 24 listas mas, que son las 24 restricciones del tipo 2.

Listado 4.9: Restricciones del tipo 2

```
1 def generate_restriction_2(self):
2     for i in range(0, 24):
3         restr_coef = [0]*5*24
4         time = (self.start+dt.timedelta(hours=i)).time()
5         if time >= dt.time(21, 30) or time <= dt.time(7, 00):
6             restr_coef[i*5] = 1
7         self.A_eq.append(restr_coef)
8         self.b_eq.append(0)
```

#### 4.4.5. Implementación de las restricciones del tipo 3

Las restricciones del tipo 3 hacen que se cumpla que la energía fotovoltaica generada no puede ser mayor que la máxima energía fotovoltaica en  $t$ , siendo  $t$  cada hora de la simulación (Véase la ecuación 4.13). Se trata de una restricción de tipo inecuación, por lo que en este caso deberán concatenarse sus valores a  $A_{ub}$  y  $b_{ub}$ . En la variable de clase *self.max\_ef\_buffer* se dispone de una lista con los 24 valores correspondientes a la energía fotovoltaica máxima de cada hora de la simulación. Cada elemento de esta lista representa la parte constante de cada restricción de este tipo, por ello, en cuanto a  $b_{ub}$  se refiere, basta con concatenar *self.max\_ef\_buffer*. En el caso de la parte de variables ( $A_{ub}$ ), al igual que en los casos anteriores se realizan 24 iteraciones correspondientes a las 24 horas de la simulación, y en cada una de ellas, la lista de 120 elementos toma el valor 1 únicamente en la posición relativa a la energía fotovoltaica, pues es la única que entra en juego en este tipo de restricción. La lista generada en cada iteración se concatena con el resto de restricciones en  $A_{ub}$ .

**Listado 4.10:** Restricciones del tipo 3

```

1 def generate_restriction_3(self):
2     for i in range(0, 24):
3         restr_coef = [0]*5*24
4         restr_coef[i*5] = 1
5         self.A_ub.append(restr_coef)
6         self.b_ub.extend(self.max_ef_buffer)

```

#### 4.4.6. Implementación de las restricciones del tipo 4

Las restricciones de tipo 4 hacen que se cumpla que la energía obtenida de la batería no puede ser mayor que el nivel de batería actual teniendo en cuenta la profundidad máxima de descarga (Véase la ecuación 4.14). Son restricciones de tipo inecuación por lo que deben modificarse  $A_{ub}$  y  $b_{ub}$ . En este caso, la restricción correspondiente a la hora 0 de la simulación debe separarse de las restantes, pues en ese punto la cantidad de carga de la batería se obtiene directamente de la variable de clase que contiene el nivel inicial de batería (*self.battery\_level*) 4.17 y en el resto de casos se obtiene mediante un conjunto de operaciones 4.18. Esto permite calcular el nivel actual de batería en la hora  $i$  a partir de la que hubo inicialmente, mediante el sumatorio de las cargas y descargas que se han realizado desde que comenzó la simulación. En el listado 4.11 se puede observar la función *generate\_restriction\_4()*, encargada de la implementación de las restricciones de tipo 4 comprendidas entre las horas 1 y 24 de la simulación. Por cada iteración, en la lista de coeficientes se coloca un uno en la posición relativa a EB, pues es la dependiente de esta restricción. Después, se realizan iteraciones desde 0 hasta la iteración anterior a la actual, para comprobar el nivel actual de batería, posicionando los valores 1 en EB y -1 en CB. Cuando la lista de coeficientes está completa en esa iteración, se añade a  $A_{ub}$ , y en  $b_{ub}$  se concatena la parte constante de este tipo de restricción, que viene a ser la diferencia entre el nivel inicial de batería y la capacidad de la misma por su profundidad de descarga.

$$EB_0 \leq initial\_level - capacity * depth \quad (4.17)$$

$$EB_i \leq initial\_level + \sum_{t=0}^{i-1} (-EB_t + CB_t) - capacity * depth \quad (4.18)$$

**Listado 4.11:** Restricciones del tipo 4

```

1 def generate_restriction_4(self):
2     for i in range(1, 24):
3         restr_coef = [0]*5*24
4         restr_coef[i*5+2] = 1
5         for j in range(0, i-1):
6             restr_coef[j*5+2] = 1
7             restr_coef[j*5+4] = -1
8         self.A_ub.append(restr_coef)
9         self.b_ub.append(self.battery_level
10            -self.battery_capacity*self.discharge_depth)

```

#### 4.4.7. Implementación de las restricciones del tipo 5

Las restricciones de tipo 5 se encargan de que el consumo para cargar la batería no pueda ser mayor que la capacidad de la misma menos el nivel restante después de  $t$  (Véase la ecuación 4.15). Las restricciones de este tipo son muy parecidas a las de tipo 4, con la diferencia de que las restricciones de tipo 4 se encargan de

regular la energía que se descarga de la batería y las restricciones de tipo 5 regulan la energía que se carga a la batería. La hora 0 de la simulación debe implementarse aparte análogamente al tipo anterior, pues el nivel actual de batería se determina en función de las cargas y descargas que se han producido desde que comenzó la simulación. En este caso la restricción de la hora 0 es muy sencilla pues tras agrupar a la izquierda de la inecuación las variables y a la derecha las constantes y ordenar las variables preservando el orden de  $f$  se obtiene la restricción 4.19. Para implementar esta restricción simplemente se debe dar valor de -1 a la posición relativa a  $EB_0$  y 1 a la posición relativa a  $CB_0$ , para después añadir a sus respectivas listas la lista de coeficientes y el valor constante de la restricción. Para el resto de restricciones de este tipo (hora 1 a 24) se usa la función `generate_restriccion_5()` cuya traza es similar a `generate_restriccion_4()` exceptuando los valores que toman las posiciones relativas a las variables dependientes de la restricción.

$$CB_0 - EB_0 \leq capacity - initial\_level - EB_0 + CB_0 \leq capacity - initial\_level \quad (4.19)$$

Listado 4.12: Restricciones del tipo 5

```

1 def generate_restriction_5(self):
2     for i in range(1, 24):
3         restr_coef = [0]*5*24
4         restr_coef[i*5+4] = 1
5         restr_coef[i*5+2] = -1
6         for j in range(0, i-1):
7             restr_coef[j*5+2] = -1
8             restr_coef[j*5+4] = 1
9         self.A_ub.append(restr_coef)
10        self.b_ub.append(self.battery_capacity - self.battery_level)

```

#### 4.4.8. Generación optimizada de energía

Una vez implementadas todas las restricciones necesarias del PSR es la hora de implementar el algoritmo linprog de Scipy. El método `optimize()` de la clase `Simulation` se encarga de esta tarea. En ella deben llamarse todos los métodos encargados de las restricciones, para así poder contener en `A_eq`, `b_eq`, `A_ub` y `b_ub` los datos de variables y constantes necesarios. Después deben determinarse los límites que pueden tomar las variables, definidos en la iteración anterior. Finalmente se efectúa el algoritmo linprog sobre todos los datos y se obtiene como respuesta un conjunto de valores que han de ser interpretados, para lo que se añaden a la clase `Simulation` las siguientes funciones auxiliares:

- **store\_result(result):** Se encarga de almacenar en un fichero los resultados obtenidos, indicando fecha de simulación, gasto económico producido y cantidad de energía de cada fuente de entrada y salida por horas. Esta información es almacenada en un fichero llamado `simulation_fecha.txt`, que sirve como reporte de la simulación. Para obtener cada valor se itera sobre la lista de valores en bruto `res.x.to_list()` separando cada valor de variable en su iteración y variable correspondiente.
- **prepare\_result(result):** Se encarga de procesar una salida a la simulación alternativa a la anterior, pues retorna los resultados utilizando el formato json. Ésto será útil cuando se haga una petición de simulación desde el servidor y deba devolverse el resultado en este formato para poder ser procesado fácilmente. Se utiliza el método `json.dumps()` para generar el objeto json a partir de un diccionario clave-valor (Véase el listado 4.13). La función `self.prepare_hours(values)` procesa la lista de valores de variables en bruto a un diccionario clave-valor.

Puesto que ya se cuenta con el esqueleto del proceso para llevar a cabo una simulación, se procede a realizar un caso de prueba del lunes 11 de Marzo de 2019.



**Listado 4.13:** Función de procesamiento del resultado a formato json

```

1 def prepare_result(self, res):
2     values = res.x.tolist()
3     data = {
4         "start" : self.start.strftime("%Y-%m-%d_%H:%M:%S"),
5         "end" : self.end.strftime("%Y-%m-%d_%H:%M:%S"),
6         "result" : res.fun,
7         "hours" : self.prepare_hours(values)
8     }
9
10    return json.dumps(data)

```

#### 4.4.9. Caso de prueba: Simulación del 11 de Marzo

Tal y cómo se explicó en la primera iteración, en el Área Cliente de Endesa es posible obtener el consumo realizado por horas de un día determinado. Esto va a permitir conocer cuál fue el consumo del 11 de Marzo y, gracias al trabajo realizado con la API e-sios [5], poder saber cuál fue la cuantía económica que supuso el consumo total de ese día, para así poder comparar con el gasto económico que se obtendrá de la simulación.

El 11 de marzo en el hogar del alumno se produjo un consumo total de 7 KWh. El precio voluntario al pequeño consumidor (PVPC) por defecto de ese día obtuvo cómo valor medio 0.12 euros, por lo tanto cómo mínimo el consumo del día supuso un gasto de 0.84 euros. El fichero de texto que contiene el consumo diario de Endesa sigue el formato mostrado en el listado 4.14 (©Endesa S.A.)

Mediante la función *read\_from\_file()* del módulo *client\_consumption* se procesa el fichero de consumo, se establecen los consumos en la unidad de KWh y se retorna una lista con los 24 consumos de las 24 del día, en este caso 11 de marzo. Esta lista será el valor que toma la variable de Simulation *self.c*. Debido a que aún no se dispone de interfaz ni servidor, para ejecutar la simulación se ha empleado el intérprete de comandos ipython3. Se debe crear un objeto simulación con los argumentos necesarios explicados a lo largo de este capítulo y que se pueden observar en la clase UML de Simulation (Figura 4.7). Tras instanciar en un objeto la simulación, se debe invocar el método *optimize()* que implementa las restricciones y ejecuta el algoritmo de programación lineal, obteniendo como salida el contenido del listado 4.15.

Esta información es difícil de interpretar estando fuera de contexto.

- *fun* es el resultado que toma la función  $f(x)$  (Función 4.16) como resultado de asignar a las variables los valores de la optimización. Dicho valor representa el **gasto económico** producido para el día simulado. Cómo se puede observar, se trata de 0.61 euros frente a los 0.84 euros que supuso el consumo del día utilizando solamente la energía de la compañía eléctrica. Se ha producido un ahorro del 27.38 % lo cuál es bastante significativo, teniendo en cuenta que el valor medio del PVPC del día 11 de marzo fue relativamente bajo con respecto al valor que suele oscilar (0.15 euros).
- *message* devuelve *feedback* sobre si la optimización ha tenido o no éxito, tomando en los campos *status* y *success* los valores 0 y True en caso de éxito; y los valores 1 y False en caso de error.
- *nit* hace referencia a las iteraciones realizadas sobre las variables para determinar el valor óptimo obtenido. En este caso se han realizado 236 iteraciones.
- *slack* contiene un array con los valores de las variables de holgura, donde cada variable de holgura corresponde con una de las restricciones de desigualdad.
- *x* contiene el array en bruto de valores de variables que han hecho posible la optimización. Esta información ha de ser tratada y clasificada para poder ser interpretada, por lo que mediante la



**Listado 4.14:** Fichero de consumo por horas de Endesa

CUPS :	ESXXXXXXXXXXXXXXXXXXXX	
Fecha :	11/03/2019	
Fecha y hora de extracción :	23/03/2019 10:57:52	
Tarifa :	No se encontró la tarifa	
Fecha	Hora	Consumo (Wh)
2019-03-11	00:00-01:00	110.0
2019-03-11	01:00-02:00	80.0
2019-03-11	02:00-03:00	166.0
2019-03-11	03:00-04:00	141.0
2019-03-11	04:00-05:00	95.0
2019-03-11	05:00-06:00	126.0
2019-03-11	06:00-07:00	186.0
2019-03-11	07:00-08:00	217.0
2019-03-11	08:00-09:00	568.0
2019-03-11	09:00-10:00	692.0
2019-03-11	10:00-11:00	280.0
2019-03-11	11:00-12:00	216.0
2019-03-11	12:00-13:00	149.0
2019-03-11	13:00-14:00	348.0
2019-03-11	14:00-15:00	677.0
2019-03-11	15:00-16:00	339.0
2019-03-11	16:00-17:00	368.0
2019-03-11	17:00-18:00	192.0
2019-03-11	18:00-19:00	202.0
2019-03-11	19:00-20:00	175.0
2019-03-11	20:00-21:00	648.0
2019-03-11	21:00-22:00	479.0
2019-03-11	22:00-23:00	318.0
2019-03-11	23:00-00:00	270.0
Total (Wh) :	7042.0	

función *store\_result()* se ha generado un fichero de reporte de simulación que contiene el valor de cada variable (EF, ER, EB, CB, CR, C y  $C_{int}$ ) en cada una de las horas de la simulación. Estos valores representan la cantidad de energía en KWh que debe tomarse en cada variable para producir la minimización del gasto económico.

Véase el Anexo A pues contiene el listado A.1 con la información procesada de la simulación. Como se puede observar, en la hora de comienzo de la simulación (0:00), la única fuente de energía es la energía de red, pues una de las restricciones implementadas es que durante la noche la energía fotovoltaica no es posible, y la batería aún no se ha cargado para permitir su extracción. Se puede observar que en algunas horas se genera más energía de la necesaria para satisfacer el consumo ( $C + C_{int}$ ). Esto es debido a que mediante la API aemet [5] se comprueba que en ese momento el precio PVPC se encuentra relativamente bajo por lo que resulta rentable importar más energía de la necesaria y almacenarla en batería para una hora en la que los precios suban. Es el caso de lo ocurrido en la hora 2:00 a.m. por ejemplo. A partir del amanecer comienza a tenerse en cuenta la energía fotovoltaica, es el caso de la hora 8:00 a.m., donde se aprovecha su potencial para importar mucha más energía de la requerida y se almacena (Se requieren 0.65 KWh pero sin embargo son generados 0.816 KWh, aprovechando la diferencia para cargar la batería). Esto permite que en horas donde por causas meteorológicas (API AE-MET [9]) no sea posible emplear energía fotovoltaica se tengan alternativas a la energía de red, o también durante la noche, como es el caso de la hora 11:00 p.m., cuando la energía fotovoltaica no es posible y la energía de red tiene un precio más alto, se obtenga la totalidad de la energía requerida de la batería (EB).

Tras esta simulación y la interpretación de resultados se han sacado las siguientes conclusiones:

**Listado 4.15:** Salida del algoritmo *linprog*

```

fun: 0.6187091110614097
message: 'Optimization terminated successfully.'
nit: 236
slack: array([ 8.16000000e-01,  8.16000000e-01,  8.16000000e-01,
               8.16000000e-01,  2.66000000e-01,  2.66000000e-01,
               2.66000000e-01,  2.66000000e-01,  0.00000000e+00,
               . . .
               5.91880000e+00,  4.68600000e+00,  1.05000000e+01,
               5.02480000e+00,  1.05000000e+01,  1.05000000e+01])
status: 0
success: True
x: array([0.00000000e+00, 1.98800000e-01, 0.00000000e+00,
          0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
          1.68800000e-01, 0.00000000e+00, 0.00000000e+00,
          . . .
          0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
          3.58800000e-01, 0.00000000e+00, 0.00000000e+00])

```

- Se ha producido un ahorro económico en la simulación del día 11 de marzo mediante el caso propuesto con respecto al caso real, a pesar que es un día donde el PVPC se encuentra relativamente bajo, por lo que el ahorro producido en un caso de simulación de la media habría sido mucho mayor.
- Mediante el empleo de distintas fuentes de energía se ha producido una eficiencia energética.
- La actuación reactiva del sistema con respecto a la información cambiante ya sea meteorológica o del mercado eléctrico permite garantizar que en cada momento la cantidad de energía en cada una de las fuentes es la cantidad óptima.

## 4.5. PERSISTENCIA DE DATOS Y CREACIÓN DE LA APLICACIÓN WEB

Una vez implementada la funcionalidad de simulación, debe pensarse en añadir una persistencia al proyecto de los elementos que intervienen. Esto es necesario y primordial si se pretende crear una aplicación web para realizar simulaciones. Puesto que se desarrollará una API usando Flask [13] que hará la función de servidor, se ha dedicado utilizar la herramienta para gestión de base de datos SQLAlchemy.

### 4.5.1. Persistencia con SQLAlchemy

SQLAlchemy [1] proporciona un kit de herramientas SQL que permiten manejar bases de datos de manera eficiente. Está formado por dos componentes:

- *Core*: Es un conjunto de herramientas de SQL que da lugar a un nivel de abstracción sobre el mismo, mediante un lenguaje que utiliza expresiones generativas en Python para expresar órdenes SQL.
- *ORM*: Se trata de un asignador relacional de objetos, es decir, permite crear una base de datos de objetos virtuales que permite manipular la información de la base de datos, a priori incompatible, como objetos utilizable por un lenguaje de programación orientada a objetos.

Mediante las consultas basadas en funciones permite ejecutar las cláusulas SQL a través de funciones y expresiones en Python. Se pueden realizar numerosas acciones como subconsultas seleccionables, insertar, actualizar, eliminar o declarar un objeto, combinaciones internas y externas sin necesidad de utilizar lenguaje SQL. El ORM permite almacenar en caché las colecciones y referencias de objetos una vez han

sido cargados, dando lugar a que no sea necesario emitir SQL en cada acceso.

SQLAlchemy puede trabajar con bases de datos de SQLite, Postgresql, MySQL, Oracle, MS-SQL, Sybase y Firebird, entre otros.

#### 4.5.2. Modelos User y Home

Se deben determinar los datos que se van a persistir en el sistema. Puesto que se tratará de una aplicación web con la que interactuarán usuarios, resulta interesante almacenarlos. Cada uno de estos usuarios realizará simulaciones del sistema. Como se vió anteriormente, cuando se instancia un objeto de la clase *Simulation*, el constructor de la misma recibe varios argumentos necesarios para llevar a cabo la simulación, de los cuales la mayoría son información acerca del hogar en el que se realizará dicha simulación (número de módulos fotovoltaicos, código de la ciudad del hogar, etc). La persistencia en base de datos de la información del hogar de cada usuario mejoraría esta situación, pues la mayoría de la información que necesita la clase *Simulation* sería proporcionada del hogar almacenado en base de datos de ese usuario. Por lo tanto, son necesarias dos tablas en la base de datos: *Users* y *Homes*, entre las cuáles existe una relación *one to one*. Este tipo de relación SQL hace que ambas tablas tengan un atributo de referencia a la otra que se conoce como **foreign key**, la cuál lo convierte en una relación bidireccional. Cada *User* tendrá un *Home* y viceversa. Para crear las tablas y mostrar la estructura lógica de cada una, así como sus limitaciones y atributos, se debe crear antes un **modelo de base de datos** que sea capaz de *mapear* cada objeto con su tabla en la base de datos.

En primer lugar se debe definir la clase *Base* mediante `sqlalchemy.ext.declarative.declarative_base()`. Ésta clase hará el papel de superclase de cada modelo. Así, cada modelo heredado de *Base* corresponde con una tabla de base de datos, cuyo nombre se encuentra en el atributo `__tablename__`. Cada objeto que se instancie la clase del modelo corresponde con un registro en base de datos. En el listado 4.16 se muestra la sintaxis de creación de un modelo. Tras `__tablename__` se declaran las columna que tendrá la tabla de ese modelo, indicando el tipo de dato que contiene y una serie de argumentos. Algunos de los que se han usado son:

- **primary\_key**: Cuando toma el valor *True* indica que ese atributo será la clave primaria de la tabla y se usará como parámetro para instanciar a la subclase de *Base*.
- **index**: Con *True* indica que se desea que dicho atributo sea indizado y permita un rápido acceso a los registros.
- **unique**: Obliga a que el atributo sea único y no permita registros con el mismo valor en esa columna de la tabla de base de datos.
- **backref**: Contiene el nombre de otra tabla en la base de datos. Este argumento permite crear una relación entre ellas.

Con la información anterior ya es posible implementar los modelos para las tablas deseadas en el caso particular de este trabajo fin de grado.

- **Modelo User**

La tabla *Users* hará referencia a los usuarios involucrados en el sistema. Existen una serie de atributos que serán propios de cada usuario y darán lugar a las columnas de la tabla:

- *name*: Representa el nombre del usuario. Este dato es de tipo String y no puede ser nulo.
- *lastname*: Representa los apellidos del usuario. Toma exactamente las mismas características que el anterior.

**Listado 4.16:** Declaración de un modelo heredado de *Base*

```

1 from sqlalchemy.ext.declarative import declarative_base
2
3 Base = declarative_base()
4
5 class <Model Name>(Base):
6     __tablename__ = <Table Name>
7     <attr name> = Column(<data type>, <arg>)
8     ...
9     ...

```

- *email*: Como su propio nombre indica almacena el correo electrónico del usuario. No puede ser nulo y debe ser único, pues este atributo identificará a cada usuario. Además es un atributo indizado, pues se realizarán consultas a base de datos a través de él.
- *password*: Contraseña definida por el usuario para acceder a su cuenta. Puesto que la contraseña es un dato sensible, debe tratarse adecuadamente. Para ello se ha hecho uso del módulo *werkzeug.security*, que se explicará en la siguiente sección de esta iteración en lo relativo a seguridad.
- *home*: Atributo apunta al registro de la tabla *Homes* que contiene el hogar de este usuario.

Véase el listado 4.17 el cuál muestra la declaración del modelo *User* heredado de base. Las funciones Integer, String, Column, relationship y declarative\_base son importadas del módulo **sqlalchemy** y permiten trabajar con abstracción sobre SQL, como se comentó anteriormente. Los métodos de clase *set\_password* y *check\_password* son usados para cambiar y comprobar la contraseña, respectivamente. Ambos llaman a funciones pertenecientes al módulo *werkzeug.security* el cuál se explicará más adelante. La función privada *\_\_repr\_\_* simplemente genera un formato para mostrar un objeto Usuario, permitiendo mostrar su nombre, apellidos e email.

**Listado 4.17:** Modelo *User*

```

1 class Users(Base):
2     __tablename__ = 'usuarios'
3     id = Column(Integer, primary_key=True)
4     name = Column(String(100), nullable=False)
5     lastname = Column(String(100), nullable=False)
6     email = Column(String(100), unique=True, index=True, ↵
7         ↵ nullable=False)
8     password_hash = Column(String(128))
9     home = relationship("Homes", uselist=False, backref="Users")
10
11     def set_password(self, password):
12         self.password_hash = generate_password_hash(password)
13
14     def check_password(self, password):
15         return check_password_hash(self.password_hash, password)
16
17     def __repr__(self):
18         return (u'<{self.__class__.__name__}: {self.id}, name={self.name}, \
19             ↵ lastname={self.lastname}, \
20             ↵ email={self.email}>'.format(self=self))

```

- **Modelo Home** La tabla *Homes* representa la casa de cada usuario. Los atributos que tendrá cada registro de esta tabla en base de datos son los siguientes:

- *city\_code*: Atributo de tipo String que hace referencia a la ciudad donde se encuentra la casa. Esto es necesario a la hora de realizar las llamadas a la API AEMET [9] pues se debe proporcionar dicho código, por lo tanto este atributo no puede ser nulo.
- *pv\_modules*: Almacena el número de módulos fotovoltaicos que tendrá el hogar del usuario, por lo tanto es un atributo de tipo Integer que no puede tomar valor nulo, ya que es un dato que determina el resultado de una simulación.
- *amortization\_years\_pv*: Almacena un entero que representa el número de años en los que el usuario desea amortizar la inversión realizada en la adquisición de los módulos fotovoltaicos. El valor de este campo determina el precio en €/KWh que tendrá la energía fotovoltaica, y por ello tampoco puede ser nulo.
- *amortization\_years\_bat*: Similar al atributo anterior pero en el contexto de la obtención de la batería. En función del valor de este atributo se calcula el precio que tiene para este usuario la obtención de energía de baterías.
- *user*: Registro de la tabla *Users* con el que mantiene una relación *one to one* y representa el usuario propietario del hogar.

Mediante el método privado `__repr__` se forma un formato para mostrar un registro de esta tabla, dando información acerca de la ciudad, número de módulos fotovoltaicos y id del usuario propietario.

Listado 4.18: Modelo *Home*

```

1 class Homes(Base):
2     __tablename__ = "homes"
3     id = Column(Integer, primary_key=True)
4     city_code = Column(String(100), nullable=False)
5     pv_modules = Column(Integer, nullable=False)
6     amortization_years_pv = Column(Integer, nullable=False)
7     amortization_years_bat = Column(Integer, nullable=False)
8     UserId = Column(Integer, ForeignKey('usuarios.id'),
9         nullable=False)
10    user = relationship("Users", backref="Homes")
11
12    def __repr__(self):
13        return (u'<{self.__class__.__name__}: {self.id}, city={self.city_code}, \
14            pv_modules={self.pv_modules}, ownerId={self.UserId}>'.format(self=self))

```

Una vez definidos los modelos se puede comenzar a realizar inserciones y consultas a estas tablas en la base de datos mediante las operaciones de SQLAlchemy, pero antes se debe comprobar el correcto funcionamiento de estos modelos. Es por esto que la etapa posterior a la implementación en el ciclo de vida del software son las **pruebas**. Mediante el framework para la implementación de casos de prueba unitarios **Nose** [10]. Véase el Anexo B, donde se hace referencia a lo relativo a pruebas en el proyecto.

Para seguir la guía de buenas prácticas se crean dos bases de datos diferenciadas. Una de ella será la base de datos de **producción**, que almacenará la información real de usuario y hogares, y será usada por la aplicación web. La segunda será la base de datos de **test**, donde se realizarán las inserciones, borrados y consultas pertinentes durante el desarrollo de los casos de prueba mencionados anteriormente para garantizar un correcto funcionamiento y coherencia entre la aplicación y la persistencia de la misma.

Al arrancar una aplicación que hace uso de la base de datos, se debe hacer una llamada al método `Base.metadata.create_all` proporcionando como argumento el objeto instanciado de `sqlalchemy.engine.base.Engine`.

Esto es necesario ya que permite crear las tablas en la base de datos proporcionada si estas no existen, y si existe, se crearán las nuevas tablas en caso de haberlas y no se eliminan los datos que existen. A partir de dicha llamada, cada objeto que se instancia de cada uno de los modelos se corresponde con un registro de su tabla correspondiente. Después de esto la base de datos estaría totalmente operable desde la aplicación. El ORM de SQLAlchemy permite realizar búsquedas relacionadas con los objetos instanciados del modelo *Base* y sus tablas relacionadas. Las operaciones se realizan en sesiones, que terminan con un *commit* que persiste los cambios realizados en la sesión en la base de datos. En el listado 4.19 se muestra una consulta para obtener el primer usuario de la tabla *Users*. Nótese que el formato de representación devuelto es el definido en la clase del modelo 4.17 mediante el método `__repr__`.

**Listado 4.19:** Consulta para obtener el primer *User*

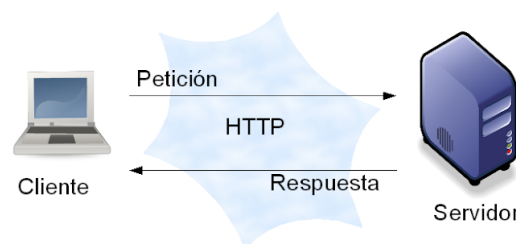
```
>>> db.session.query(Users).first()
<Users: 1, name= Pablo Palomino Gomez, email= pablo@eoptimizer.com>
```

Para integrar la persistencia que se ha incorporado al funcionamiento actual, debe refactorizarse la clase *Simulation*. El constructor pasa de recibir un argumento por atributo de clase a recibir únicamente tres atributos: *home*, *user* y *date*. Toda la información necesaria para una simulación que antes se obtenía a través del fichero de constantes del proyecto (y no entendía de usuarios) ahora se puede obtener de los objetos *User* y *Home* del usuario que realiza la simulación. El tercer parámetro (*date*) hace referencia a un objeto de tipo *datetime* correspondiente al día en el que se realiza la simulación.

A partir de esta integración, se permite realizar simulaciones dependientes de un usuario, que obtendrá un valor totalmente distinto a si otro usuario realiza la simulación el mismo día, pues cada uno de ellos tiene unos parámetros en su hogar que determinan el resultado, como son el número de módulos fotovoltaicos, la ciudad donde reside o el precio al que obtiene las energías fotovoltaica y de batería, ya que cada usuario define el periodo en el que desea amortizar lo invertido en cada fuente mediante ganancias de esa fuente.

### 4.5.3. Implementación de una aplicación Flask

El siguiente paso tras la integración de persistencia y la implementación de la lógica o *backend* es crear una aplicación web siguiendo la arquitectura **cliente - servidor** [18]. En esta arquitectura, cada una de las máquinas que realiza una demanda de información al sistema toma el rol de cliente, y la que responde a estas demandas toma el rol de servidor (Figura 4.8). Esto permitirá que exista un servidor el cuál realiza simulaciones a petición de clientes, de los que previamente se ha almacenado la información necesaria en base de datos mediante un registro.



**Figura 4.8:** Esquema Cliente - Servidor

El nombre que tomará dicha aplicación web será **eOptimizer**, haciendo referencia a su objetivo principal: realizar simulaciones por medio de una optimización. En la figura 4.9 se muestra el logo creado

para la aplicación. Dicho logo ha sido diseñado por el propio alumno y se inspira en la eficiencia energética que se consigue mediante el sistema que se propone.

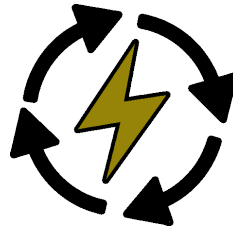


Figura 4.9: Logo de eOptimizer

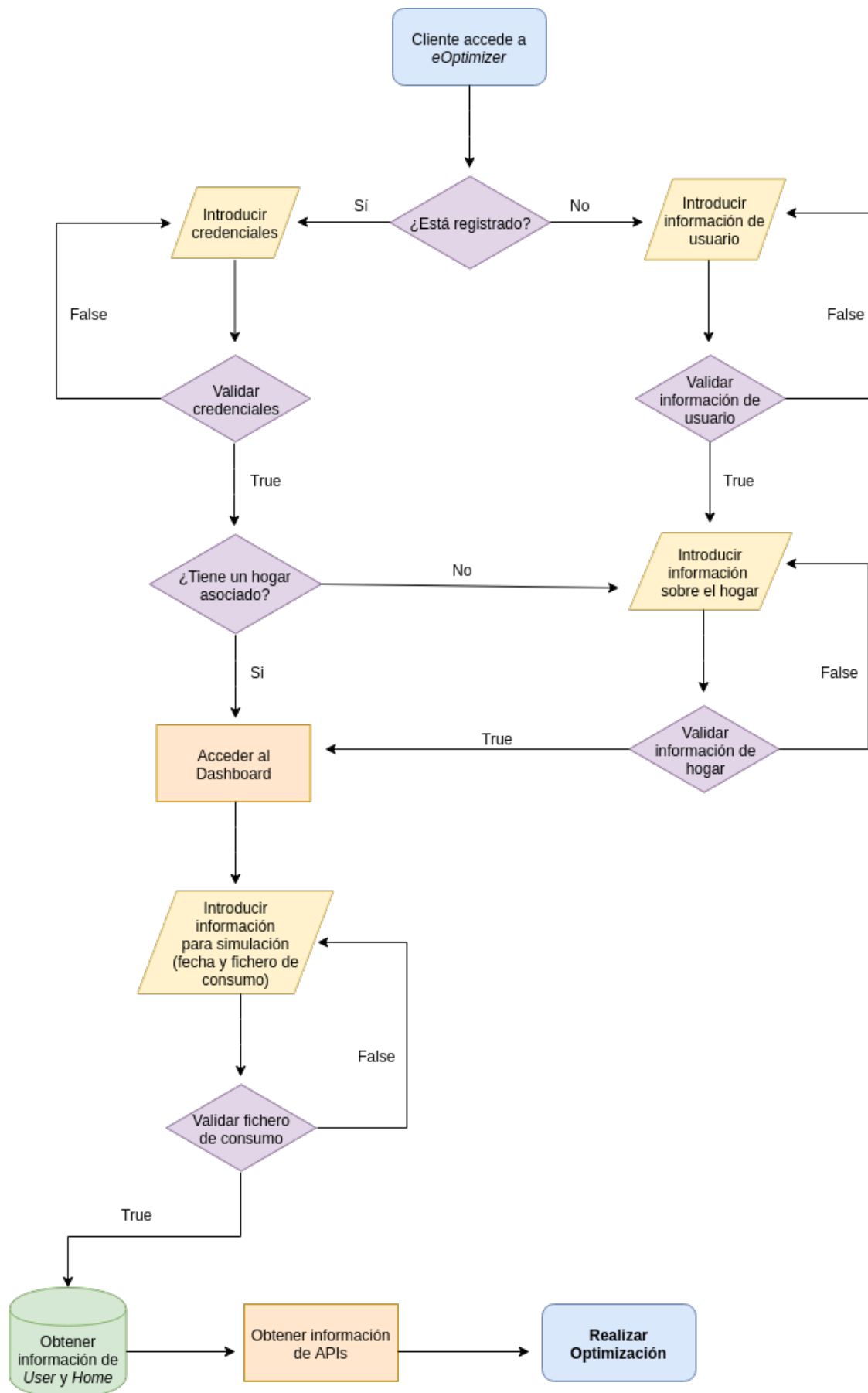
Antes de entrar en la implementación de la aplicación debe definirse correctamente el funcionamiento que ha de tener. Esto implica determinar el flujo posible de interacción de un cliente con la aplicación web, algo que a la hora de implementar captura de errores y restricciones facilitará mucho la complejidad. Para ello ha de construirse un **diagrama de flujo**. Este tipo de diagrama describe un sistema informático, y tiene como objetivo planificar, estudiar y diseñar procesos complejos o algoritmos. De entre todas las funcionalidades que pueden implementar los diagramas de flujo, deben mencionarse:

- Mostrar la ejecución del código de un software.
- Facilitar la comprensión de la estructura y funcionalidad de una aplicación web.
- Explicar visualmente cómo un usuario puede navegar por una página web.

Para crear un diagrama de flujo debe tenerse en cuenta el alcance del sistema. En el caso de este trabajo fin de grado, un usuario (rol de cliente) interactuará con la aplicación web (servidor) con el objetivo de realizar una **simulación** del consumo eléctrico de un día determinado que habría tenido su hogar mediante el sistema propuesto, comparándolo frente a su situación actual, por tanto la entrada al diagrama de flujo debe ser el hecho de acceder a la aplicación web, y la salida será la información pertinente a la simulación realizada.

Lo primero que debe hacer un cliente es identificarse, pues en el servidor debe tenerse en cuenta el usuario de la base de datos que está interactuando con el sistema para realizar la simulación, ya que necesita la información asociada. Por ello, el usuario deberá introducir sus credenciales para poder acceder. Si un usuario nunca antes ha accedido a eOptimizer, deberá registrarse. Una vez haya introducido la información de usuario para su registro, esta se insertará en base de datos y si todo va correctamente, el siguiente paso es introducir la información asociada a su hogar, que será empleada para realizar sus simulaciones. Cuando se haya introducido se insertará en base de datos, teniendo persistida toda la información necesaria del usuario en curso, por lo que se permitiría el acceso al índice o *dashboard* de la web. Desde allí un usuario podrá realizar simulaciones, introduciendo los datos necesarios para la misma, como son la fecha que desea simular y el fichero de consumo de Endesa que hubo en su hogar ese día. Si esta información es correcta, el *backend* realizará la optimización habiendo obtenido previamente la información de las APIs AEMET y Esios relativa al día de simulación, mostrando los resultados obtenidos al cliente. En la figura 4.10 se muestra el diagrama de flujo de la aplicación que permite visualizar todo el funcionamiento de manera mas sencilla. Nótese que en color azul se muestran la entrada y salida del flujo, en color morado se muestran las decisiones, en color amarillo los datos de entrada, en color naranja los procesos y en color verde el acceso a base de datos.



**Figura 4.10:** Diagrama de flujo de eOptimizer



Una vez definido el flujo que debe seguir la aplicación web, el siguiente paso de esta iteración es la implementación del servidor. Para ello se hace uso de **Flask** [20]. Flask es un pequeño framework -autodenominado microframework- para desarrollo web con Python. Sigue una filosofía minimalista por lo que con un pequeño número de líneas de código permite crear aplicaciones web. Este framework tiene dos dependencias principales:

- **Werkzeug**: Proporciona los componentes necesarios de *routing*, *debugging* y WSGI (*Web Server Gateway Interface*).
- **Jinja2**: Proporciona las herramientas para la integración de *templates* html en un servidor Flask.

Desafortunadamente Flask no cuenta con soporte nativo para integración con base de datos, pero como se mencionó anteriormente, SQLAlchemy se integra perfectamente con este framework de desarrollo web, por lo que se dispone de todos los componentes necesarios para la creación de la aplicación web. Las buenas prácticas de Flask recomiendan usar **virtualenv** para la instalación de las dependencias de una aplicación web. Un virtualenv [12] es una copia privada del intérprete de Python donde se pueden integrar paquetes y librerías sin afectar al entorno del sistema, de manera que permite tener un entorno privado por aplicación web en la misma máquina donde las dependencias de cada aplicación están organizadas en sendos estancos sin afectarse entre sí. Para este trabajo fin de grado se ha creado un virtualenv donde mediante el gestor de paquetes de Python3 **pip3** se han instalado las dependencias necesarias del proyecto. Estas dependencias se han ido mencionando a lo largo de este capítulo (Scipy, SQLAlchemy, Flask, Werkzeug, security, etc) y se encuentran en el fichero *requirements.txt*.

En un servidor Flask deben definirse las rutas o *endpoints* que existirán en la aplicación web. La petición de un cliente web a un *endpoint* disparará la ejecución de la función asociada a esa ruta, lo que lo convierte en un framework sencillo pero a la vez muy potente. En la tabla 4.2 se muestran los *endpoints* de la aplicación eOptimizer. Nótese que cada *endpoint* puede recibir un determinado tipo de operación http.

<b>Endpoint</b>	<b>Operación HTTP</b>	<b>Descripción</b>
'/'	GET	Representa el <i>index</i> de la aplicación web. Su función será redirigir al formulario de inicio de sesión si no hay usuario identificado o al <i>dashboard</i> si hay usuario identificado.
'/login'	POST	Petición de inicio de sesión con las credenciales de usuario.
'/logout'	GET	Petición de cierre de sesión del usuario identificado.
'/signup'	POST	Petición de registro de usuario con la información necesaria.
'/add-home'	POST	Petición de registro de hogar del usuario identificado proporcionando la información necesaria.
'/simulation'	POST	Petición de realizar una simulación en el hogar del usuario identificado proporcionando la información necesaria.
errorhandler(404)	GET	Hook de captura de errores cuya labor será redirigir a la página de error cuando se realice una petición a un <i>endpoint</i> que no existe.

**Tabla 4.2:** *Endpoints* de la aplicación web

Ha de implementarse correctamente el control de sesiones y la autenticación de usuarios. Mediante el módulo del que se habló en el apartado anterior, **Werkzeug.security** [11] se controla lo relativo a seguridad. Dicho módulo contiene funciones (denominadas *helpers*) para manejo de contraseñas como

son *generate\_password\_hash* y *check\_password\_hash*. Nótese como ambas son utilizadas en la clase del modelo *User* (listado 4.17). La primera recibe como argumento una cadena de texto que es la contraseña introducida por el nuevo usuario. Esta función genera un *hash* encriptado que almacena en base de datos, de tal forma que la información sensible queda protegida y no es vulnerable. La segunda función es llamada cuando se desea comprobar la veracidad de unas credenciales. Recibe una cadena de texto y comprueba si el *hash* almacenado tras ser descriptado coincide con dicha cadena de texto. Estas funciones son de gran ayuda a la hora de implementar el inicio de sesión en eOptimizer, pues se comprueba si el email introducido (atributo indizado) existe en base de datos y si es así, se comprueba si la contraseña introducida coincide con la de dicho usuario haciendo uso de esta herramienta. En lo referente al control de sesiones, Flask cuenta con el módulo **session**, el cuál guarda la información de sesión de un cliente web determinado durante cierto tiempo en el atributo 'logged\_in'. Por lo tanto, la lógica a seguir en el inicio de sesión pasa por comprobar si el usuario con el email introducido existe. Si existe, se comprueba la contraseña introducida mediante el método *check\_password*. Si es correcta, el usuario habría sido autenticado, lo que se traduce en almacenar el objeto *User* en una variable global **currentUser** y dar valor true a *session['logged\_in']*. Por lo tanto, existen tres salidas posibles de login: usuario correcto (éxito), usuario no existe (fallo) y contraseña incorrecta (fallo). Todas las operaciones posteriores que requieran de autenticación comprobarán los valores de esas dos variables.

Para realizar una simulación, existen varios condicionales que comprueban posibles errores o incompatibilidades antes de efectuarla. Si todo es correcto, se instancia un objeto de la clase *Simulation* proporcionándole los objetos *Home* y *User* relativos al usuario autenticado además de la fecha que se desea simular y el fichero de consumo obtenido de Endesa. El valor de retorno de llamar a la función *optimize* de ese objeto *Simulation* es un json que contiene los valores obtenidos en parejas clave-valor.

Los valores de retorno de los métodos asociados a cada *endpoint* llaman a la función auxiliar de Flask **render\_template**. Esta función emplea las utilidades de Jinja2 y muestra al cliente la plantilla html definida en la función. Los posibles errores son acumulados en una lista *errors*, la cuál es enviada como argumento a **render\_template**. Esto es debido a una funcionalidad que integra Jinja2, que permite insertar código en una plantilla html, pudiendo mostrar variables, implementar condicionales, bucles, etc. De esta forma el json obtenido de la simulación se consigue mostrar correctamente en el *frontend*. En el listado 4.20 se muestra el proceso de mostrar los posibles errores producidos en el formulario de inicio de sesión haciendo uso de esta funcionalidad.

**Listado 4.20:** Muestreo de errores en el formulario de inicio de sesión

```

1 {% if error %}
2 <div class="alert alert-danger" role="alert">
3   {{ error }}
4 </div>
5 {% endif %}

```

Al levantar un servidor Flask, este recibe la configuración pertinente de un fichero de configuración. En este trabajo fin de grado se proporciona mediante *prod\_config* o *test\_config*. Cada uno de ellos levanta una instancia diferente de la aplicación. El primero enlaza el servidor de Flask con la base de datos de producción, y el segundo con la base de datos de test. Para poner en funcionamiento una instancia del servidor se ha implementado la clase *run*, mostrada en el listado 4.21. En él se maneja la app mediante el módulo *Manager* de *flask\_script* que permite manejar un servidor de desarrollo de una aplicación Flask. La configuración de producción mencionada anteriormente es aplicada a app, para después instanciar la base de datos y ejecutar el *create\_all* del que se habló anteriormente. A partir de ahí el servidor está corriendo y listo para recibir peticiones.

**Listado 4.21:** Módulo *run* para arrancar el servidor

```
#!/usr/bin/python3
# -*- coding: utf-8; mode: python -*-

from flask_script import Manager
from flask_sqlalchemy import SQLAlchemy
from app import app, db
from config import prod_config
from models import Base

# Deploy to production
manager = Manager(app)
app.config.from_object(prod_config)
db = SQLAlchemy(app)
Base.metadata.create_all(bind=db.engine)

if __name__ == '__main__':
    manager.run()
```

#### 4.5.4. Frontend de la aplicación web

Para la implementación del *frontend* y desarrollo web se ha utilizado la librería **Bootstrap** [2]. Esta librería proporciona un conjunto de herramientas para crear sitios web *responsives* y livianos. Entre estas herramientas se encuentran clases html predefinidas, funciones de Javascript, etc. Esto ha permitido una base sobre la que trabajar para construir el sitio web. Se han implementado las siguientes plantillas html:

- **Base:** Contiene el encabezado y el pie de página de la web y es heredada en el resto de plantillas.
- **Login:** Vista que muestra el formulario de inicio de sesión y el de nuevo usuario en eOptimizer. Da como resultado la ejecución del *endpoint* asociado `/login` o `/signup`, devuelto condicionalmente mediante las herramientas de Jinja2 explicadas anteriormente.
- **New\_home:** Vista que muestra el formulario de introducción de datos asociados al hogar del usuario. Como resultado se ejecuta el método del *endpoint* asociado `/add-home`, que inserta en base de datos la información proporcionada.
- **Dashboard:** Vista principal de eOptimizer una vez autenticado. Cuenta con una tabla de información acerca de la información del hogar correspondiente y con un formulario para realizar simulaciones (Figura 4.11).
- **Simulation:** Vista de reporte de una simulación. Una vez es realizada la optimización, la función *render\_template* carga esta vista procesando la información del json obtenido (Figura 4.12, donde se muestra esta vista de la simulación del 16 de Abril de 2019). Contiene tres funcionalidades: mostrar la tabla de distribución de la energía, mostrar la tabla de precios que se dieron el día simulado, y por último descargar el fichero de reporte de simulación.
- **Not\_found:** Vista que se muestra cuando el *hook* errorhandler captura una ruta no existente. Informa del error y da opción de volver al *dashboard*.

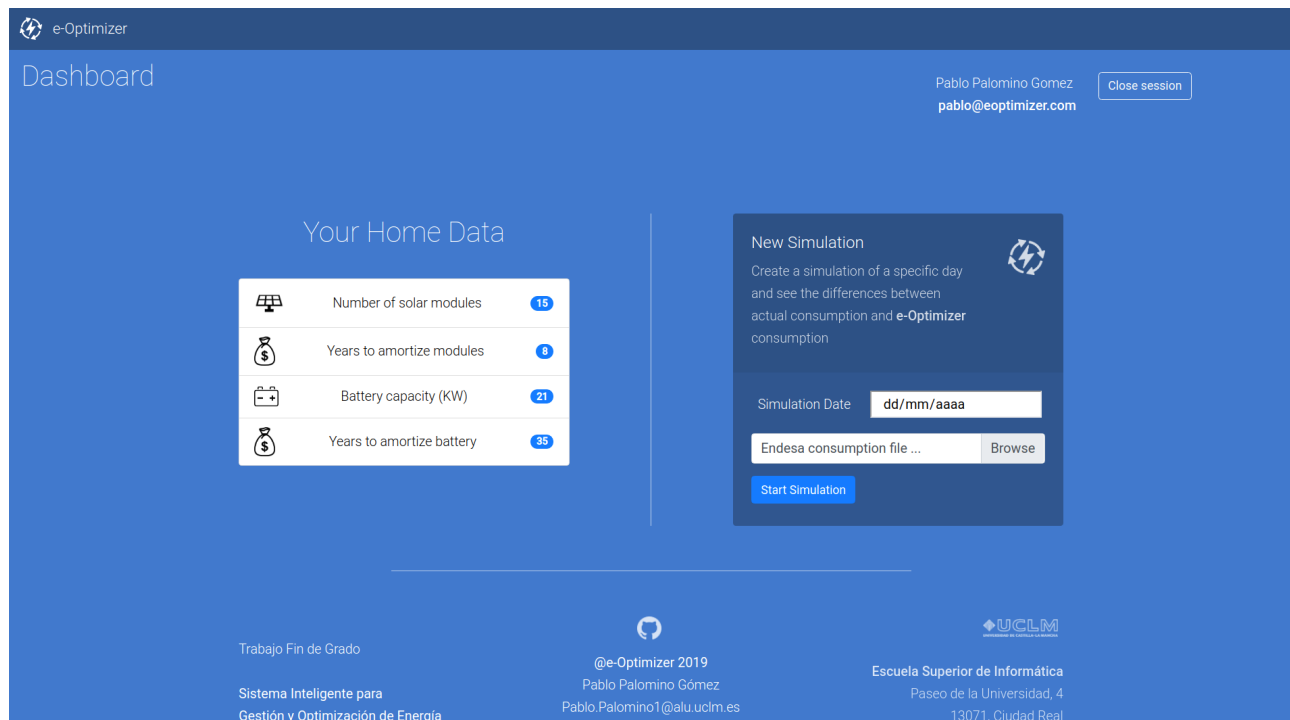


Figura 4.11: Vista *dashboard* de la aplicación

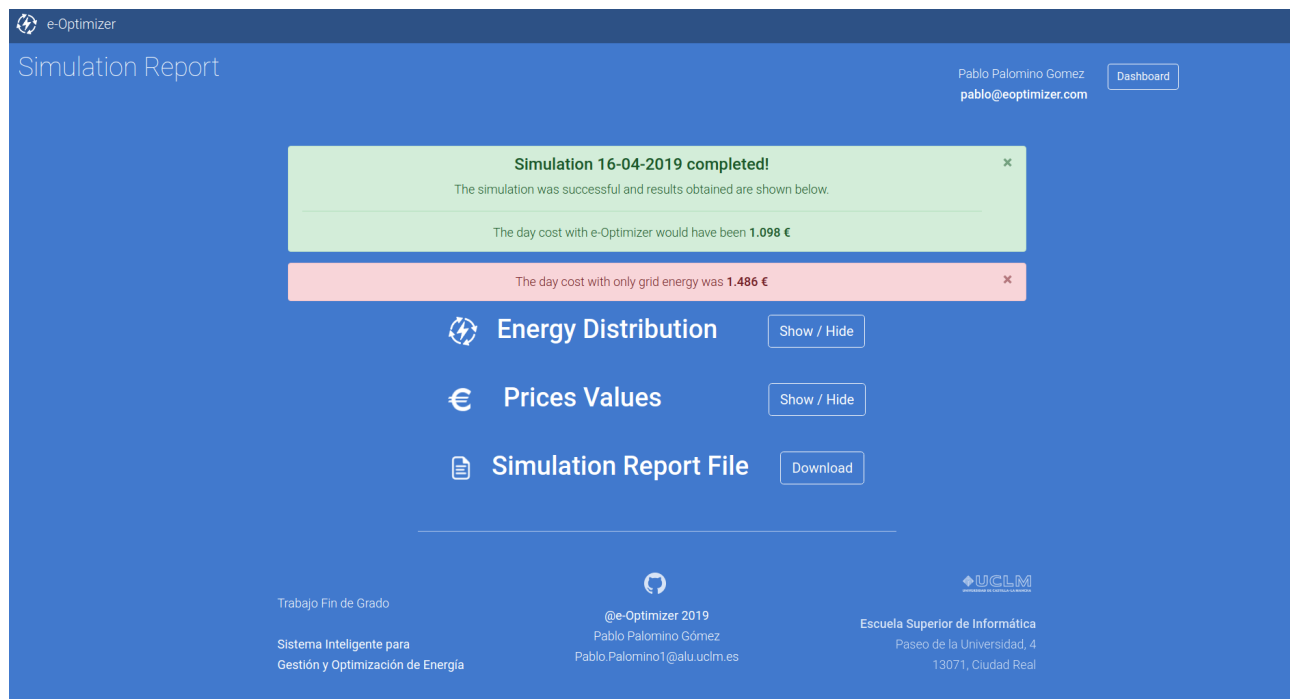


Figura 4.12: Vista *simulation report* de la aplicación

En el listado A.1 del Anexo A se mostró el resultado de una simulación durante la iteración 4. Dicha información se corresponde con el fichero de reporte de simulación, que a partir de ahora se puede descargar en la vista Simulation (Figura 4.12). Además la información de la simulación es procesada por horas en la tabla **energy distribution**, que mediante funciones de javascript muestra barras de distribución de la energía por horas, con el fin de facilitar la comprensión del usuario y hacerlo más visible. Véase la figura 4.13, donde se muestra la tabla en las horas comprendidas entre la 13:00 y las 23:00 de la

optimización del día 16 de Abril de 2019.

13:00	572.4 Wh	0.0 Wh	675.6 Wh	1204.0 Wh	0.0 Wh	0.0 Wh	44.0 Wh	EF	EB	C	B
14:00	572.4 Wh	0.0 Wh	171.6 Wh	700.0 Wh	0.0 Wh	0.0 Wh	44.0 Wh	EF	EB	C	CB
15:00	572.4 Wh	112.6 Wh	0.0 Wh	641.0 Wh	0.0 Wh	0.0 Wh	44.0 Wh	EF	ER	C	CB
16:00	572.4 Wh	2030.0 Wh	0.0 Wh	878.0 Wh	0.0 Wh	1680.4 Wh	44.0 Wh	EF	ER	C	CB
17:00	572.4 Wh	0.0 Wh	0.0 Wh	335.0 Wh	0.0 Wh	193.4 Wh	44.0 Wh	EF	ER	C	CB
18:00	272.4 Wh	333.6 Wh	0.0 Wh	562.0 Wh	0.0 Wh	0.0 Wh	44.0 Wh	ER	ER	C	CB
19:00	272.4 Wh	1094.6 Wh	0.0 Wh	1323.0 Wh	0.0 Wh	0.0 Wh	44.0 Wh	EF	ER	C	B
20:00	272.4 Wh	0.0 Wh	1956.6 Wh	2185.0 Wh	0.0 Wh	0.0 Wh	44.0 Wh	EF	EB	C	B
21:00	272.4 Wh	0.0 Wh	360.6 Wh	589.0 Wh	0.0 Wh	0.0 Wh	44.0 Wh	EF	EB	C	CB
22:00	0.0 Wh	0.0 Wh	532.0 Wh	488.0 Wh	0.0 Wh	0.0 Wh	44.0 Wh	EB	EB	C	CB
23:00	0.0 Wh	209.6 Wh	171.4 Wh	337.0 Wh	0.0 Wh	0.0 Wh	44.0 Wh	ER	EB	C	CB

Figura 4.13: Tabla *energy distribution* del 16/04 entre 13:00-23:00

Como salida de esta iteración se obtiene la versión totalmente funcional de la aplicación web integrada con el *backend* para optimizar simulaciones. Actualmente solo es posible ejecutarlo en la máquina local, siendo dependiente del software y dependencias propios de la misma. En la siguiente iteración se buscará la integración en la nube de la aplicación.

Los tests relativos a la aplicación web (*routing* y vistas) que demuestran la correcta funcionalidad son explicados en el Anexo B.

## 4.6. MIGRACIÓN DE LA APLICACIÓN A LA NUBE MEDIANTE UNA IAAS

El objetivo de esta iteración será la integración de la aplicación web **eOptimizer** en la nube.

La computación en la nube (*cloud computing*) permite ubicuidad en el acceso a un conjunto de recursos compartido, como pueden ser servidores, bases de datos, contenedores, almacenamiento, servicios, etc. Estos recursos pueden ser aprovisionados o liberados con facilidad y bajo demanda, teniendo una facturación por uso y siendo accedidos a través de la red, lo que da lugar a que esta tendencia cada vez esté mas en auge. Los recursos a disposición del usuario son ilimitados, teniendo a su disposición arquitecturas de computación realmente complejas y modernas sin gran esfuerzo.

Antes del *cloud computing* el hecho de crear un servidor que funcionase 24 horas al día, 7 días a la semana era algo costoso, ya que no solo implica la inversión de la máquina si no que conlleva unos costes de mantenimiento (alimentación y red constantes, mejora de componentes desfasados a lo largos del tiempo, mantenimiento técnico, etc). Con esta nueva tendencia, es posible contratar el servicio deseado facturando únicamente por uso sin necesidad de todos los gastos anteriores.

El primer paso es analizar que recursos en la nube son necesarios para la aplicación web de este trabajo fin de grado. Existen un gran tipo de recursos disponibles de distintos proveedores:

- Instancias
- Bases de Datos SQL

- Bases de Datos NoSQL
- Almacenamiento en la nube
- IP virtuales SSL

Por un lado es interesante que la persistencia del sistema cuente con un recurso propio, para lo que sería necesario una base de datos SQL, pues se utiliza el framework SQLAlchemy que trabaja sobre bases de datos SQL. El proveedor elegido para la instancia de este recurso ha sido la plataforma **IBM Cloud**



**Figura 4.14:** Logo de la plataforma IBM Cloud

Además la aplicación web de este trabajo necesita de un servidor que ha de estar funcionando constantemente, para el que sería necesario una instancia. El proveedor elegido para este recurso ha sido **Amazon Web Services (AWS)**



**Figura 4.15:** Logo de Amazon Web Services

# **CONCLUSIONES**

---

En este capítulo se realizará un juicio crítico y discusión sobre los resultados obtenidos. Si es pertinente deberá incluir información sobre trabajos derivados como publicaciones o ponencias, así como trabajos futuros, solo si estos están planificados en el momento en que se redacta el texto. Además incluirá obligatoriamente la explicación de cómo el trabajo realizado satisface las competencias de la tecnología específica cursada.





# REPORTE DE SIMULACIÓN 11/04/2019

## Listado A.1: Reporte de simulación 11/04

```
-----  
----- Simulacion Completada -----  
-----  
Inicio: 2019-03-11 00:00:00  
Fin: 2019-03-12 00:00:00  
Gasto: 0.6187091110614097 €  
  
Valores a las 00:00 11/03/19  
- EF = 0.0 Kwh  
- ER = 0.19879999999999853 Kwh  
- EB = 0.0 Kwh  
- CR = 0.0 Kwh  
- CB = 0.0 Kwh  
- Consumo del Hogar = 0.11 Kwh  
- Consumo del Sistema = 0.0888 Kwh  
-----  
Valores a las 01:00 11/03/19  
- EF = 0.0 Kwh  
- ER = 0.16880000000000628 Kwh  
- EB = 0.0 Kwh  
- CR = 0.0 Kwh  
- CB = 5.329070518200751e-15 Kwh  
- Consumo del Hogar = 0.08 Kwh  
- Consumo del Sistema = 0.0888 Kwh  
-----  
Valores a las 02:00 11/03/19  
- EF = 0.0 Kwh  
- ER = 0.25479999999999947 Kwh  
- EB = 0.0 Kwh  
- CR = 0.0 Kwh  
- CB = 0.0 Kwh  
- Consumo del Hogar = 0.166 Kwh  
- Consumo del Sistema = 0.0888 Kwh  
-----  
Valores a las 03:00 11/03/19  
- EF = 0.0 Kwh  
- ER = 0.2297999999999938 Kwh  
- EB = 0.0 Kwh  
- CR = 0.0 Kwh  
- CB = 0.0 Kwh  
- Consumo del Hogar = 0.141 Kwh  
- Consumo del Sistema = 0.0888 Kwh  
-----  
Valores a las 04:00 11/03/19  
- EF = 0.0 Kwh  
- ER = 1.460599999999996 Kwh
```

- EB = 0.0 Kwh  
- CR = 0.0 Kwh  
- CB = 1.2767999999999962 Kwh  
- Consumo del Hogar = 0.095 Kwh  
- Consumo del Sistema = 0.0888 Kwh

-----  
Valores a las 05:00 11/03/19

- EF = 0.0 Kwh  
- ER = 0.21480000000000388 Kwh  
- EB = 0.0 Kwh  
- CR = 0.0 Kwh  
- CB = 0.0 Kwh  
- Consumo del Hogar = 0.126 Kwh  
- Consumo del Sistema = 0.0888 Kwh

-----  
Valores a las 06:00 11/03/19

- EF = 0.0 Kwh  
- ER = 0.0 Kwh  
- EB = 0.27479999999999905 Kwh  
- CR = 0.0 Kwh  
- CB = 0.0 Kwh  
- Consumo del Hogar = 0.186 Kwh  
- Consumo del Sistema = 0.0888 Kwh

-----  
Valores a las 07:00 11/03/19

- EF = 0.0 Kwh  
- ER = 0.0 Kwh  
- EB = 0.30580000000000014 Kwh  
- CR = 0.0 Kwh  
- CB = 0.0 Kwh  
- Consumo del Hogar = 0.217 Kwh  
- Consumo del Sistema = 0.0888 Kwh

-----  
Valores a las 08:00 11/03/19

- EF = 0.816 Kwh  
- ER = 0.0 Kwh  
- EB = 0.0 Kwh  
- CR = 0.0 Kwh  
- CB = 0.15920000000000023 Kwh  
- Consumo del Hogar = 0.568 Kwh  
- Consumo del Sistema = 0.0888 Kwh

-----  
Valores a las 09:00 11/03/19

- EF = 0.266 Kwh  
- ER = 0.0 Kwh  
- EB = 0.51479999999999939 Kwh  
- CR = 0.0 Kwh  
- CB = 0.0 Kwh  
- Consumo del Hogar = 0.692 Kwh  
- Consumo del Sistema = 0.0888 Kwh

-----  
Valores a las 10:00 11/03/19

- EF = 0.266 Kwh  
- ER = 0.0 Kwh  
- EB = 0.10279999999999845 Kwh  
- CR = 0.0 Kwh  
- CB = 0.0 Kwh  
- Consumo del Hogar = 0.28 Kwh  
- Consumo del Sistema = 0.0888 Kwh

-----  
Valores a las 11:00 11/03/19

- EF = 1.816 Kwh  
- ER = 0.0 Kwh

```
- EB = 0.0 Kwh
- CR = 0.0 Kwh
- CB = 1.511199999999997 Kwh
- Consumo del Hogar = 0.216 Kwh
- Consumo del Sistema = 0.0888 Kwh
```

-----

Valores a las 12:00 11/03/19

```
- EF = 0.0 Kwh
- ER = 0.0 Kwh
- EB = 0.2378 Kwh
- CR = 0.0 Kwh
- CB = 0.0 Kwh
- Consumo del Hogar = 0.149 Kwh
- Consumo del Sistema = 0.0888 Kwh
```

-----

Valores a las 13:00 11/03/19

```
- EF = 0.0 Kwh
- ER = 0.0 Kwh
- EB = 0.43680000000000163 Kwh
- CR = 0.0 Kwh
- CB = 0.0 Kwh
- Consumo del Hogar = 0.348 Kwh
- Consumo del Sistema = 0.0888 Kwh
```

-----

Valores a las 14:00 11/03/19

```
- EF = 0.816 Kwh
- ER = 0.0 Kwh
- EB = 0.0 Kwh
- CR = 0.0 Kwh
- CB = 0.05020000000000202 Kwh
- Consumo del Hogar = 0.677 Kwh
- Consumo del Sistema = 0.0888 Kwh
```

-----

Valores a las 15:00 11/03/19

```
- EF = 3.816 Kwh
- ER = 0.0 Kwh
- EB = 0.0 Kwh
- CR = 0.0 Kwh
- CB = 3.3882000000000057 Kwh
- Consumo del Hogar = 0.339 Kwh
- Consumo del Sistema = 0.0888 Kwh
```

-----

Valores a las 16:00 11/03/19

```
- EF = 0.8160000000000001 Kwh
- ER = 0.0 Kwh
- EB = 0.0 Kwh
- CR = 0.0 Kwh
- CB = 0.3592000000000013 Kwh
- Consumo del Hogar = 0.368 Kwh
- Consumo del Sistema = 0.0888 Kwh
```

-----

Valores a las 17:00 11/03/19

```
- EF = 0.816 Kwh
- ER = 0.0 Kwh
- EB = 0.0 Kwh
- CR = 0.0 Kwh
- CB = 0.535199999999997 Kwh
- Consumo del Hogar = 0.192 Kwh
- Consumo del Sistema = 0.0888 Kwh
```

-----

Valores a las 18:00 11/03/19

```
- EF = 0.0 Kwh
- ER = 0.0 Kwh
```

```

- EB = 0.2907999999999973 Kwh
- CR = 0.0 Kwh
- CB = 0.0 Kwh
- Consumo del Hogar = 0.202 Kwh
- Consumo del Sistema = 0.0888 Kwh
-----

```

Valores a las 19:00 11/03/19

```

- EF = 0.8159999999999998 Kwh
- ER = 0.0 Kwh
- EB = 0.0 Kwh
- CR = 0.14539999999999864 Kwh
- CB = 0.4068000000000005 Kwh
- Consumo del Hogar = 0.175 Kwh
- Consumo del Sistema = 0.0888 Kwh
-----

```

Valores a las 20:00 11/03/19

```

- EF = 0.266 Kwh
- ER = 0.0 Kwh
- EB = 5.116400000000001 Kwh
- CR = 4.645600000000003 Kwh
- CB = 0.0 Kwh
- Consumo del Hogar = 0.648 Kwh
- Consumo del Sistema = 0.0888 Kwh
-----

```

Valores a las 21:00 11/03/19

```

- EF = 0.816 Kwh
- ER = 0.0 Kwh
- EB = 0.047999999999998266 Kwh
- CR = 0.2962000000000007 Kwh
- CB = 0.0 Kwh
- Consumo del Hogar = 0.479 Kwh
- Consumo del Sistema = 0.0888 Kwh
-----

```

Valores a las 22:00 11/03/19

```

- EF = 0.0 Kwh
- ER = 0.0 Kwh
- EB = 0.4068000000000005 Kwh
- CR = 0.0 Kwh
- CB = 0.0 Kwh
- Consumo del Hogar = 0.318 Kwh
- Consumo del Sistema = 0.0888 Kwh
-----

```

Valores a las 23:00 11/03/19

```

- EF = 0.0 Kwh
- ER = 0.0 Kwh
- EB = 0.3587999999999987 Kwh
- CR = 0.0 Kwh
- CB = 0.0 Kwh
- Consumo del Hogar = 0.27 Kwh
- Consumo del Sistema = 0.0888 Kwh
-----

```

**Bibliografía** Esta sección, que si se prefiere puede titularse «Referencias», incluirá un listado por orden alfabético (primer apellido del primer autor) con todas las obras en que se ha basado para la realización del TFG en las que se especificará: autor/es, título, editorial y año de publicación. Solo se incluirán en esta sección las referencias bibliográficas que hayan sido citadas en el documento. Todas las fuentes consultadas no citadas en el documento deberían incluirse en una sección opcional denominada «Material de consulta», aunque preferiblemente estas deberían incluirse como referencias en notas a pie de página a lo largo del documento.

Se usará método de citación numérico con el número de la referencia empleada entre corchetes. La

cita podrá incluir el número de página concreto de la referencia que desea citarse. Debe tenerse en cuenta que el uso correcto de la citación implica que debe quedar claro para el lector cuál es el texto, material o idea citado. Las obras referenciadas sin mención explícita o implícita al material concreto citado deberían considerarse material de consulta y por tanto ser agrupados como «Material de consulta» distinguiéndolas claramente de aquellas otras en las que si se recurre a la citación.

Cuando se desee incluir referencias a páginas genéricas de la Web sin mención expresa a un artículo con título y autor definido, dichas referencias podrán hacerse como notas al pie de página o como un apartado dedicado a las «Direcciones de Internet».

Todo el material ajeno deberá ser citado convenientemente sin contravenir los términos de las licencias de uso y distribución de dicho material. Esto se extiende al uso de diagramas y fotografías. El incumplimiento de la legislación vigente en materia de protección de la propiedad intelectual es responsabilidad exclusiva del autor del trabajo independientemente de la cesión de derechos que este haya convenido. De este modo será responsable legal ante cualquier acción judicial derivada del incumplimiento de los preceptos aplicables. Así mismo ante dicha circunstancia los órganos académicos se reservan el derecho a imponer al autor la sanción administrativa que se estime pertinente.

**Índice temático** Este índice es opcional y se empleará como índice para encontrar los temas tratados en el trabajo. Se organizará de modo alfabético indicando el número de página(s) en el que se aborda el tema concreto señalado.



# TESTS DE LA APLICACIÓN UTILIZANDO NOSE

Las pruebas o *tests* son la etapa del ciclo de vida del software que consiste en la implementación de casos de prueba unitarios sobre cada una de las funcionalidades de una aplicación. Completar los tests con éxito en una aplicación implica:

- La aplicación es utilizable.
- Se logra obtener el resultado esperado o definido para cada funcionalidad de la aplicación.
- Existe control y manejo de errores.
- La ejecución se realiza en un tiempo aceptable.
- Los requisitos definidos durante el diseño y desarrollo han sido implementados correctamente.

Se ha utilizado el framework **Nose** para implementación de casos de prueba en Python. Se implementan tests que comprueban cada uno de los módulos que componen el sistema. En el directorio **test** se encuentran todos los ficheros relativos a las pruebas. (Véase el *tree* del listado [B.1](#))

**Listado B.1:** Estructura del directorio *test*

```
--- test
|  |-- factories
|  |  |
|  |  |-- factory_models.py
|  |-- tests_api.py
|  |-- tests_app.py
|  |-- tests_models.py
```

La factoría de modelos (*factory\_models*) se encarga de generar un usuario o hogar a petición de los tests, lo que agiliza muchísimo el proceso y fomenta la reutilización de código.

Los tests de API se encargan de comprobar la funcionalidad de las APIs empleadas en el sistema (Esios y AEMET). Las peticiones son probadas ejecutando las funciones de cada módulo y comprobando si el resultado obtenido es un buffer de 24 posiciones en caso de probar el éxito, o un buffer vacío en caso de probar el fallo.

Los tests de modelos comprueban la corrección de cada uno de los modelos *User* y *Home* realizando inserciones correctas, erróneas, borrados, y demás operaciones en la base de datos de test. Para ello se configura la aplicación en modo test mediante el fichero de configuración *test\_config*.

El módulo de test *tests\_app* tiene una gran importancia, pues contiene los casos de prueba unitarios referentes al *routing* y vistas de la aplicación web. Para el primer tipo, se realizan casos de prueba cuyos

*asserts* comprueba los códigos http de retorno de cada una de las peticiones posibles a los *endpoints* de la aplicación. Para el segundo tipo, los test de vistas, se comprueba si la información mostrada en la respuesta a una petición contiene lo esperado mediante *assertIn*.

Cada módulo de test tiene dos funciones auxiliares *setUp* y *tearDown* cuyo objetivo es preparar la aplicación para efectuar los siguientes casos de prueba, y regenerar la base de datos de test y limpiar la basura generada en los tests.

Nose cuenta con una opción **–with-coverage** que permite comprobar la cobertura de código alcanzada en cada uno de los ficheros del proyecto. Esto es importante pues si existe una gran proporción de líneas de código probadas es mucho más difícil que un error inesperado ocurra. En la tabla B.1 se muestra la cobertura alcanzada mediante la ejecución de los tests en el proyecto.

<i>Fichero</i>	<b>Cobertura de código</b>
app.py	85 %
models.py	94 %
simulation.py	99 %
config/prod_config.py	100 %
config/project_constants.py	100 %
helpers/api_aemet.py	98 %
helpers/api_esios.py	100 %

**Tabla B.1:** Cobertura alcanzada en los tests del proyecto



# BIBLIOGRAFÍA

---

## FUENTES ONLINE

- [1] Michael Bayer. *SQLAlchemy library*. 2006. URL: <https://www.sqlalchemy.org/features.html> (visited on 04/24/2019).
- [2] Bootstrap Community. *About Bootstrap*. 2010. URL: <https://getbootstrap.com/docs/4.3/about/overview/> (visited on 05/15/2019).
- [3] Ministerio de Energía. *Real Decreto 900/2015*. 2015. URL: <https://www.boe.es/buscar/act.php?id=BOE-A-2015-10927&p=20181006&tn=0> (visitado 26-03-2019).
- [4] Red Eléctrica de España. *Precio Voluntario para el Pequeño Consumidor*. 2014. URL: <https://www.ree.es/es/actividades/operacion-del-sistema-electrico/precio-voluntario-pequeno-consumidor-pvpc> (visitado 25-03-2019).
- [5] Red Eléctrica de España S.A. *API Rest e-sios Docs*. URL: <https://www.esios.ree.es/es/pagina/api> (visitado 15-01-2019).
- [6] IBM. *IBM Cloud Docs*. 2016. URL: <https://console.bluemix.net/docs/overview/ibm-cloud-platform.html> (visited on 12/28/2018).
- [7] Kennethreitz. *Requests: HTTP for Humans*. URL: <http://docs.python-requests.org/en/master> (visited on 03/28/2019).
- [8] Python Standard Library. *datetime — Basic date and time types*. URL: <https://docs.python.org/3/library/datetime.html> (visited on 03/28/2019).
- [9] Agencia Estatal de Meteorología. *API AEMET OpenData Docs*. URL: <https://opendata.aemet.es/centrodedescargas/inicio> (visitado 27-12-2018).
- [10] Comunidad Nose. *Nose. Extends unittest to make testing easier*. 2015. URL: <https://nose.readthedocs.io/en/latest/> (visited on 05/11/2019).
- [11] Pallets. *Werkzeug Utilities*. 2007. URL: <https://werkzeug.palletsprojects.com/en/0.15.x/utils/#module-werkzeug.security> (visited on 05/12/2019).
- [12] Pypa. *Virtualenv Docs*. 2007. URL: <https://virtualenv.pypa.io/en/stable/> (visited on 04/12/2019).
- [13] Armin Ronacher. *Flask Docs*. 2015. URL: <http://flask.pocoo.org/docs/1.0/> (visited on 12/20/2018).
- [14] Comunidad Scipy. *Módulo Scipy Linprog*. 2015. URL: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linprog.html> (visited on 04/18/2019).
- [15] Comunidad Scipy. *Módulo Scipy.optimize*. 2015. URL: <https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html> (visited on 04/18/2019).
- [16] Comunidad Scipy. *Scipy eco-system*. 2015. URL: <https://scipy.org/about.html> (visited on 04/16/2019).

## FUENTES NO ONLINE

- [17] Juan Manuel García Sánchez. *Gestión de la eficiencia energética : cálculo del consumo, indicadores y mejora*. AENOR, D.L., 2012.
- [18] John Goerzen. *Foundations of Python Network Programming*. 1st ed. Apress, 2004.
- [19] Carlos González Morcillo. *Lógica Difusa, una introducción práctica. Técnicas de Softcomputing*. Escuela Superior de Informática, 2011.
- [20] Miguel Grinberg. *Flask Web Development*. 1st ed. O'REILLY, 2014.
- [21] Robert C. Martin James Newkirk. *La Programación Extrema en la Práctica*. ADDISON WESLEY, Pearson Educación S.A., 2002.
- [22] Narendra Paul Loomba. *Linear Programming: An introductory analysis*. McGraw-Hill, 1964.
- [23] Perpiñán O. *Diseño de Sistemas Fotovoltaicos*. ProgenSA, 2012.
- [24] Norving P. Russell S. *Inteligencia Artificial, un enfoque moderno*. 2.<sup>a</sup> ed. Prentice Hall, 2006.
- [25] L.A. Zadeh. *Fuzzy Set*. Information and Control, 1965.
- [26] L.A. Zadeh. *Outline of a new approach to the analysis of complex system*. IEEE Transaction on System Man and Cybernetics, 1973.