

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

Grado en Ingeniería Eléctrica

PROYECTO Fin de Grado

TÍTULO	SIMULACIÓN DE CONCENTRADORES CILINDROPARABÓLICOS CON PYTHON 3
AUTOR	FRANCISCO JOSÉ MUNUERA PÉREZ
DIRECTOR	RUBÉN BARBERO FRESNO
CODIRECTOR	
PONENTE	
DEPARTAMENTO	INGENIERÍA ENERGÉTICA

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS INDUSTRIALES

RESUMEN - ABSTRACT

Palabras clave: Energía solar de concentración; Python; Concentrador Cilindroparabólico; CCP

En este trabajo se desarrolla una librería software en Python 3 en la que se implementan las clases necesarias para el modelado de un campo solar de colectores cilindroparabólicos (CCP).

Se sigue una metodología basada en la Programación Orientada a Objetos, en la que cada sistema físico se modela mediante una Clase. El principal modelo matemático empleado para el calculo del rendimiento del concentrador solar es el Modelo de 4º Orden propuesto en [1].

A partir del código desarrollado se muestran algunos ejemplos de aplicación para el análisis paramétrico de CCP y el análisis del rendimiento de un campo solar real.

Keywords: Concentrated Solar Energy; Python; Parabolic Trough Collectors; PTC

A software library in Python 3 is developed for the modelling of solar field of parabolic-trough collectors (PTC)

A methodology based on Object Oriented Programming (OOP) is followed and each physical system is modeled through a Class. The main mathematical model used to calculate the performance of the solar concentrator is the 4th Order Model proposed in [1].

Some application of the code are shown, including parametric analysis of PTC and the analysis of the performance of a real solar field.

AGRADECIMIENTOS

Agradezco encarecidamente al profesor Rubén Barbero su dedicación, cercanía y paciencia.

También quisiera mostrar mi agradecimiento y admiración hacia todas aquellas personas que comparten sus conocimientos y herramientas de forma desinteresada. Resulta sorprendente y esperanzador descubrir la cantidad de contenidos, explicaciones, tutoriales y herramientas de código que muchos hombres y mujeres comparten públicamente, ayudando así al progreso de toda la comunidad.



Esta obra se encuentra sujeta a la licencia Creative Commons
Reconocimiento - No Comercial - Sin Obra Derivada

ÍNDICE GENERAL

1. INTRODUCCIÓN	1
1.1. Objetivo	1
1.2. Estructura de la memoria	3
1.3. Concentradores cilindroparabólicos	3
1.3.1. El concentrador cilindroparabólico	4
1.3.2. El tubo absorbedor o receptor	6
1.3.3. El sistema de seguimiento	7
1.3.4. El fluido caloportador	8
2. DESCRIPCIÓN DE LOS MODELOS TEÓRICOS UTILIZADOS	9
2.1. Modelo de rendimiento térmico de 4º Orden para caracterización de un sistema de captación solar	9
2.2. Modelo de Primer Orden	14
2.3. Modelo simplificado	16
2.4. Aplicabilidad de los modelos	16
3. MODELADO DEL CAMPO SOLAR	18
3.1. Metodología seguida para el modelado del campo solar	18
3.2. Sistemas físicos y Clases para el modelado del campo solar	18
3.2.1. Clases derivadas de Model: Clase ModelBarbero4thGrade, ModelBarbero1stGrade y ModelBarberoSimplified	21
3.2.2. Clase HCE para modelado del Heat Collector Element	23
3.2.3. Clase SCA, para el modelado del Solar Collector Assembly	30
3.2.4. Clase Loop	31
3.2.5. Clase Subfield, modelado del subcampo	33

3.2.6. Clase <code>Solarfield</code> , modelado del campo solar	33
3.2.7. Clase <code>Fluid</code> y sus clases hijas, <code>FluidCoolProp</code> y <code>FluidTabular</code>	34
3.2.8. Clases <code>Weather</code> , <code>FieldData</code> y <code>TableData</code>	38
3.2.9. Clase <code>Site</code>	39
3.3. Procedimiento para realizar una simulación	40
3.4. Verificación por comparación con otra herramienta de simulación	44
3.4.1. Configuración de la simulación para comparación con SAM	45
3.4.2. Guía para la configuración de la simulación en Python.	49
3.4.3. Resultados de la verificación	62
4. APLICACIONES DEL CÓDIGO DE SIMULACIÓN.	69
4.1. Aplicación para el análisis paramétrico	69
4.1.1. Rendimiento del HCE en función de la radiación normal directa, <i>DNI</i>	69
4.1.2. Rendimiento del HCE en función de la temperatura de entrada	73
4.1.3. Rendimiento del HCE en función del flujo de radiación absorbido, \dot{q}_{abs}''	76
4.1.4. Simulación con los diferentes modelos teóricos.	78
4.1.5. Simulación cambiando el tamaño de la malla de integración	83
4.2. Análisis de los datos de generación de una planta solar termoeléctrica real	88
5. CONCLUSIONES Y TRABAJO FUTURO	96
BIBLIOGRAFÍA	100
GLOSARIO	104
CÓDIGO FUENTE: CSENERGY.PY	105
CÓDIGO FUENTE: INTERFACE.PY	

ÍNDICE DE FIGURAS

1.1	Tecnologías solares de concentración	4
1.2	Lazo de concentradores cilindroparabólicos	5
1.3	Vista de perfil de un SCA	6
2.1	Esquema del receptor empleado para el modelo	10
3.1	Esquema relacional de las Clases HCE, SCA y Loop	20
3.2	Esquema relacional de las Clases SolarField, Subfield y Loop	21
3.3	Parámetros físicos del <i>Dowtherm-A</i> en estado de líquido saturado	36
3.4	Esquema de operación global de una simulación de un campo solar mediante la clase <i>SolarFieldSimulation</i>	43
3.5	Esquema desarrollado del bloque de <i>procesamiento del lazo</i> para las simulaciones tipo <i>simulation</i> y <i>benchmark</i>	44
3.6	Configuración SAM. Selección del archivo de datos meteorológicos . .	46
3.7	Configuración SAM. Campo solar	47
3.8	Configuración SAM. Configuración del SCA SenerTrough I a partir del modelo EuroTrough ET150	48
3.9	Configuración SAM. Configuración del HCE UVAC 3 con vacío	48
3.10	Configuración del tipo de simulación	50
3.11	Configuración de la simulación del campo solar	52
3.12	Proceso de asignación de <i>tags</i> a variables de entrada/salida	53
3.13	Ventana con la lista de <i>tags</i> disponibles	54
3.14	Lista de asignaciones <i>tag-variable</i> completa	54

3.15	Configuración de la simulación. Selección y configuración del fluido caloportador	55
3.16	Configuración de la simulación. Selección del modelo de SCA y configuración	56
3.17	Configuración de la simulación. Selección del modelo de HCE y configuración	57
3.18	Consola de Python mostrando el inicio de una simulación	61
3.19	Caudal mÁsico calculado por SAM y por el cÓdigo Python para el dÍa 17/07/2007	63
3.20	Temperatura de entrada y de salida calculadas por SAM y por el cÓdigo Python para el dÍa 17/07/2007	64
3.21	Potencia tÉrmica calculada por SAM y por el cÓdigo Python para el dÍa 17/07/2007	64
3.22	Caudal mÁsico calculado por SAM y por el cÓdigo Python para el dÍa 17/06/2007	66
3.23	Temperatura de entrada y de salida calculadas por SAM y por el cÓdigo Python para el dÍa 17/06/2007	66
3.24	Potencia tÉrmica calculada por SAM y por el cÓdigo Python para el dÍa 17/06/2007	67
4.1	Rendimiento tÉrmico en funciÓn de DNI para diferentes modelos de HCE	72
4.2	Rendimiento tÉrmico en funciÓn de la temperatura de entrada del HTF para diferentes modelos de HCE	76
4.3	Rendimiento tÉrmico en funciÓn del flujo de radiaciÓn absorbido para diferentes modelos de HCE	77
4.4	Temperaturas de salida obtenidas con los tres modelos	80
4.5	Caudales de salida obtenidos con los tres modelos en un dÍa de condiciones estables	80

4.6	Potencia térmica obtenida con cada uno de los tres modelos en un día de condiciones estables	81
4.7	Rendimiento calculado con cada modelo para un tamaño de malla de integración de 4,05 m	84
4.8	Rendimiento calculado con cada modelo para un tamaño de malla de integración de 48,60 m	84
4.9	Rendimiento calculado con cada modelo para un tamaño de malla de integración de 72,90 m	85
4.10	Rendimiento calculado con cada modelo para un tamaño de malla de integración de 145,80 m	85
4.11	Rendimiento calculado para diferentes tamaños de malla de integración	86
4.12	Desviación, para simulaciones con el Modelo de 4º Orden, según diferentes tamaños de malla	87
4.13	Desviación, para simulaciones con el Modelo de 1 ^{er} Orden, según diferentes tamaños de malla	87
4.14	Desviación, para simulaciones con el Modelo de Simplificado, según diferentes tamaños de malla	88
4.15	Vista aérea de las centrales Aste 1A y Aste 1B	89
4.16	Reflectividad registrada durante el mantenimiento	90
4.17	Temperaturas de operación reales y simuladas en un día de condiciones estables	92
4.18	Potencia térmica real y simulada en un día de condiciones estables	92
4.19	Caudal real y simulado en un día de condiciones estables	93
4.20	Potencia térmica real y simulada en función de DNI	94

ÍNDICE DE TABLAS

3.1	Coeficientes polinómicos para el fluido caloportador <i>Dowtherm-A</i>	37
3.2	Coeficientes polinómicos para el fluido caloportador <i>Therminol VP-1</i>	38
3.3	Configuración del SCA modelo SenerTrough I de Sener	47
3.4	Configuración del HCE modelo UVAC 3 de Solel	48
3.5	Resultados globales anuales para las simulaciones con SAM y Python . .	62
3.6	Resultados de las simulaciones en un día de condiciones estables	65
3.7	Resultados de las simulaciones en un día de condiciones inestables	67
4.1	Constantes del modelo de emisividad equivalente para cada uno de los receptores seleccionados	69
4.2	Rendimiento en función de la radiación normal incidente para distintos modelos de HCE	72
4.3	Rendimiento en función de la temperatura de entrada del HTF para diferentes modelos de HCE	74
4.4	Rendimiento en función del flujo de radiación absorbido y para cada modelo teórico	77
4.5	Temperaturas obtenidas en la simulación con cada modelo teórico en un día de condiciones estables	78
4.6	Caudales obtenidos en la simulación para cada modelo en un día de condiciones estables	81
4.7	Potencia térmica calculada en la simulación de cada modelo en un día de condiciones estables	82

1. INTRODUCCIÓN

1.1. Objetivo

El propósito principal de este Trabajo Final de Grado (TFG) es profundizar en el conocimiento del funcionamiento de un campo solar de concentradores cilindroparabólicos (CCP) mediante el desarrollo de una herramienta de simulación programada en Python 3. Para ello, se ha partido del modelo teórico desarrollado en su Tesis Doctoral por el profesor de la Universidad Nacional de Educación a Distancia (UNED), el Dr. Rubén Barbero Fresno y se ha empleado una metodología basada en el paradigma de la programación orientada a objetos (POO). El desarrollo de este trabajo ha supuesto un reto personal por la necesidad de adquirir habilidades en el manejo de diferentes herramientas antes desconocidas para mí, como el lenguaje de programación Python 3 [2] y sus diferentes librerías para el cálculo científico, concretamente para la resolución de ecuaciones mediante métodos numéricos (*Numpy*, [3]) y el tratamiento masivo de datos (*Pandas*, [4]). Finalmente, todo el código se encuentra publicado y accesible a través de *GitHub* y se pretende crear una versión interactiva online mediante *Notebook Jupyter* (en proceso).

El propósito inicial era crear un código con un limitado número de Clases para el modelado del HCE pero, según el trabajo iba avanzando, se amplió para cubrir la necesidad de modelar también los SCA, el lazo completo, subcampos y campo solar completo. También el fluido caloportador ha merecido especial atención con el fin de que sea posible aprovechar la existencia de librerías software como *CoolProp* [5] para la obtención de las propiedades de los fluidos. Los mecanismos de entrada y salida de datos son lo suficientemente flexibles como para permitir la lectura de archivos meteorológicos en formato *TMY* y *CSV*, para lo que se aprovecha parte las librerías desarrolladas para el código de simulación de sistemas fotovoltaicos, *pvlib-python*, [6].

Al mismo tiempo, se vio la necesidad de que las configuraciones de estos elementos permaneciesen guardas en ficheros que facilitasen su recuperación posteriormente, a modo de base de datos, pero que también fuesen fácilmente editables, con el fin de que el usuarios pudiera configurar sus propios modelos con un simple editor de texto. Por este

motivo se optó por el formato *JSON* para guardar la configuración de las simulaciones y de los sistemas que conforman el campo solar.

Para que el usuario no tenga que partir de cero a la hora de crear una configuración, se ha desarrollado también una sencilla interfaz gráfica con *TkInter* [7] que ayuda en el proceso de creación del archivo de configuración *JSON*. Esta interfaz permite cargar una configuración ya creada o comenzar a partir de una plantilla. El usuario puede guardar la nueva configuración con el nombre que desee para usarla posteriormente.

Los resultados obtenidos han sido verificados comparándolos con los obtenidos mediante System Advisor Model (SAM, [8]). Los cálculos relativos a la posición solar y óptica geométrica presentan valores casi idénticos, con diferencias despreciables achacables a los redondeos propios del cálculo digital. Los cálculos relativos a rendimientos térmicos y otra variables termodinámicas presentan algunas diferencias, como es de esperar por el hecho de emplear modelos diferentes. Pero en todo caso, los resultados obtenidos son compatibles con los ofrecidos por SAM, teniendo en cuenta que en este proyecto solo se aborda la simulación del sistema del campo solar, el único sistema dentro del alcance del modelo teórico de partida.

Finalmente, también se ha realizado un comparativa con los datos de generación de una planta termosolar real. En este caso, debido a las peculiaridades de funcionamiento de la planta real, cuyo campo solar está muy sobredimensionado, solo podríamos comparar la potencia térmica de salida del campo en aquellos momentos en los que supiésemos que no se están aplicando limitaciones de generación al campo. No obstante, sí hemos podido emplear nuestro código para estimar la magnitud de este sobredimensionado y exceso de energía térmica disponible en el campo.

La metodología seguida permite que en el futuro este proyecto pueda ser ampliado de forma sistemática con la incorporación de nuevas clases de objetos que aprovechen los métodos de entrada y salida ya programados para interactuar con ellos. En todo caso, se han empleado valores de rendimientos estimados para los principales subsistemas de planta con el fin de ofrecer una estimación de la energía eléctrica finalmente vertida a la red.

1.2. Estructura de la memoria

Esta memoria se estructura en 5 capítulos y un anexo con el código fuente.

En este primer capítulo se describen los objetivos del TFG y se ofrece una introducción a las características de los concentradores cilindroparabólicos. El segundo capítulo presenta el modelo teórico de partida y se detallan las ecuaciones de los modelos para el cálculo del rendimiento térmico de los tubos absorbedores que más adelante serán modelados. El tercer capítulo aborda el modelado de los diferentes sistemas necesarios para la caracterización del campo solar y se procede a la verificación por comparación con otra herramienta de simulación. En el cuarto capítulo se desarrollan, a modo de ejemplo de aplicación, algunos análisis paramétricos de diferente tipo. También se contrastan los resultados de la simulación del campo solar con los datos disponibles de una planta termosolar real. Finalmente, en el quinto capítulo se presentan algunas conclusiones y propuestas de desarrollo futuro.

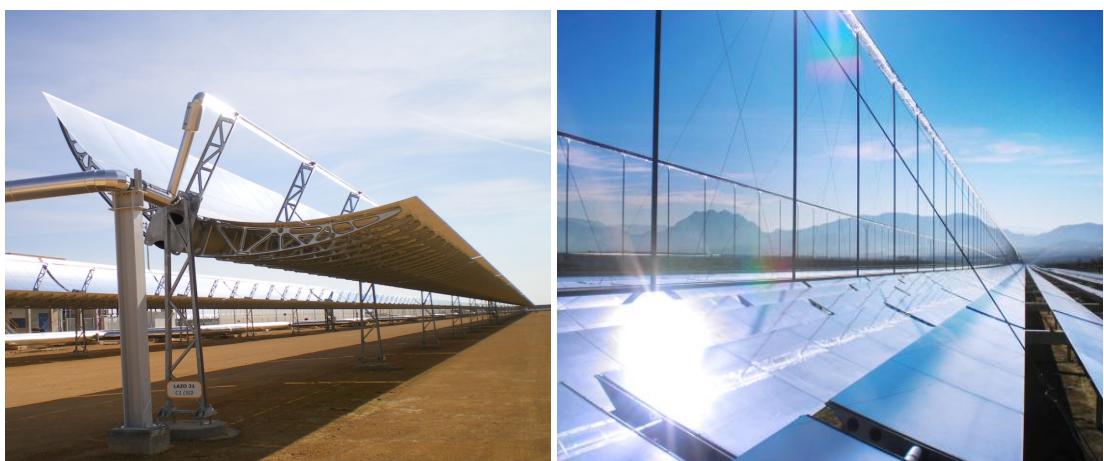
1.3. Concentradores cilindroparabólicos

Existen principalmente cuatro tecnologías para el aprovechamiento de la radiación solar directa en sistemas térmicos: concentrador cilindro-parabólico (CCP), central de torre, concentrador Fresnel y disco parabólico. La tecnología de CCP es la que cuenta en la actualidad con una mayor madurez y gran número de centrales en operación y en construcción en todo el mundo [9]. La tecnología de torre central ha crecido en los últimos años, pero todavía está lejos de la madurez de la tecnología de CCP. En España, actualmente hay medio centenar de centrales CCP en operación [10], alguna de ellas desde hace más de una década, quedando probado que el estado del arte y la madurez de la tecnología garantizan el correcto funcionamiento del sistema. Este tipo de sistema presenta un alto grado de replicabilidad, modularidad y aprovechamiento del terreno. Desde el punto de vista económico, esta tecnología también resulta muy favorable ya que los costes de inversión y operación han sido comercialmente probados, al menos, para los sistemas termoeléctricos.



(a) Central Solar de Torre. Fuente [10]

(b) Concentradores de Disco Parabólico. Fuente [10]



(c) Colector Cilindro-Parabólico. Fuente: CST Aste 1A (d) Concentrador de tipo Fresnel. Fuente [10]

Fig. 1.1. Tecnologías solares de concentración

En un CCP pueden distinguirse cuatro elementos principales: el reflector o concentrador, el tubo absorbedor, el sistema de seguimiento y el fluido caloportador.

1.3.1. El concentrador cilindroparabólico

Un CCP consiste en una superficie a modo canal de sección parabólica que refleja la radiación solar directa concentrándola sobre un tubo absorbedor colocado en la línea focal del parabolóide. Dentro de los diferentes sistemas de concentración solar pertenece al grupo de los concentradores lineales, al igual que los sistemas de concentración tipo Fresnel y al contrario que los sistemas de concentración de torre central o de discos parabólicos, en cuyo caso estaríamos hablando de sistemas de concentración puntuales. Por

el tubo absorbedor se puede hacer circular algún fluido que se calentará debido a la radiación incidente sobre el tubo. Se trata de una transformación directa de radiación solar en energía térmica con una buena eficiencia y que puede alcanzar temperaturas de hasta 675 K con los aceites sintéticos actuales. Se podrían alcanzar temperaturas mayores con sales fundidas o gases, pero el empleo de estos fluidos caloportadores todavía no cuenta con experiencias comerciales a gran escala.



Fig. 1.2. Perspectiva de un lazo completo. Se indica el sentido del recorrido del HTF en el lazo, desde la tubería de entrada hasta la de salida. Fuente: CST Aste 1A

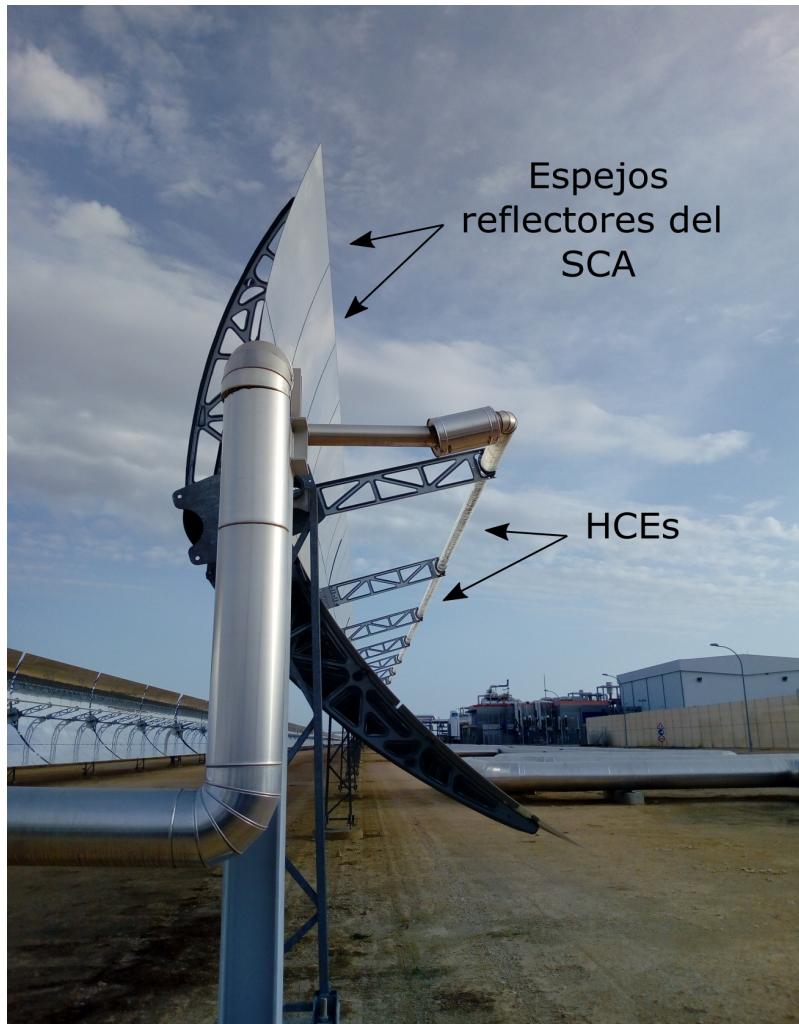


Fig. 1.3. Vista de perfil de un SCA. Pueden distinguirse algunos HCEs y los brazos de soporte. Fuente: CST Aste 1A

1.3.2. El tubo absorbedor o receptor

A lo largo del eje focal del concentrador se instala una conducción por la que circula un fluido caloportador o transmisor del calor (HTF por sus siglas en inglés, Heat Transfer Fluid). Esta conducción está compuesta en realidad por una serie de elementos tubulares denominados Heat Collector Element, HCE. Los HCE consisten en tubo de acero con una envolvente de vidrio de tal forma que en el proceso de fabricación se ha dejado extraído el aire que queda entre ambos (región anular o *annulus*). De esta forma se reducen las pérdidas de calor por convección a través de la región anular. La soldadura vidrio-metal y unos elementos denominados *getters* que absorben, hasta cierto punto, algunas moléculas que puedan filtrarse a la región anular durante la vida de operación del HCE permiten que

éste cuente con pérdidas reducidas de calor mientras no se produzca la rotura del vidrio o la saturación de dichos *getters*.

1.3.3. El sistema de seguimiento

Para que se produzca la concentración de la radiación solar incidente ésta debe ser perpendicular al eje que pasa por el foco y la base de la parábola. La primera consecuencia es que solo puede aprovecharse plenamente la componente normal de la radiación solar incidente (DNI). Dado que el Sol varía su posición relativa al concentrador continuamente, el conjunto reflector-tubo absorbedor está montado sobre una estructura que pueda girar sobre un eje con el fin de seguir la trayectoria solar a lo largo del día. Salvo instalaciones especiales en laboratorios, no se emplean seguidores a dos ejes pues la complejidad de los colectores con movimiento basado en dos ejes es tal que no permite su rentabilidad debido a los costes de mantenimiento. Por tanto, el sistema de seguimiento más empleado consiste en mover la estructura del colector con un grado de libertad en torno a un eje que, en las plantas solares cuyo objetivo es maximizar el vertido anual de energía eléctrica a la red, cuenta con una orientación Norte-Sur, lo cual lleva a que exista una importante diferencia entre la generación en los meses de verano y los meses de invierno, siendo mayor en los primeros. Si lo que se persigue es obtener una producción más estable a lo largo del año, la orientación más adecuada del eje sería Este-Oeste.

La rotación del colector requiere un mecanismo de accionamiento, eléctrico o hidráulico, dependiendo en muchos casos de las dimensiones y el peso de los elementos del colector. Para abaratar costes se suele emplear un mismo mecanismo para mover varios módulos.

El control del movimiento se puede llevar a cabo de forma autónoma, en el propio colector, dotándolo de algún dispositivo para detectar la posición del Sol en el cielo. Otra opción es emplear algoritmos matemáticos que calculan la posición del Sol para cada momento del día, en cualquier día del año. Una vez calculada la posición solar, se mueve el colector hasta colocarlo correctamente orientado. Este método requiere algún sistema para conocer la posición exacta del colector. Lo normal es emplear un codificador angular. Dado que el colector solar se encuentra en movimiento, las conexiones del tubo absorbedor con las tuberías de entrada y salida de éste deben permitir el giro en los puntos de

unión. Para esto se emplean conexiones flexibles y juntas rotativas combinadas adecuadamente. El coste de estos elementos es también elevado por lo que la elección de una configuración adecuada puede suponer un importante ahorro de costes en la instalación y el mantenimiento.

1.3.4. El fluido caloportador

Actualmente, el rango de temperatura de trabajo con colectores cilindro parabólicos es de 425 K a 675 K. El agua desmineralizada es una buena opción para temperaturas inferiores a los 450 K. A mayor temperatura es preferible el aceite sintético debido a que no aumenta tanto su presión. Si el campo solar está acoplado a un ciclo termodinámico para generación de energía eléctrica, se podría maximizar el rendimiento alcanzando temperaturas más altas, pero no hay ningún fluido que pueda dar unas prestaciones tecnológicas adecuadas a temperaturas superiores sin un aumento significativo del coste. Actualmente se están desarrollando algunas experiencias con sales fundidas y están en desarrollo sistemas de generación directa de vapor (emplean directamente el agua como fluido caloportador). Para temperaturas inferiores hay otros colectores más económicos (colectores de placa plana y colectores de tubo de vacío).

2. DESCRIPCIÓN DE LOS MODELOS TEÓRICOS UTILIZADOS

2.1. Modelo de rendimiento térmico de 4º Orden para caracterización de un sistema de captación solar

Hay dos tendencias principales en el desarrollo de modelos analíticos que describan el comportamiento de un captador solar: por un lado, modelos empíricos o semiempíricos basados en resultados de ensayos y, en mayor o menor medida, aproximaciones para captadores concretos (véase por ejemplo [11], [12], [13] o [14]) y, por otro lado, modelos teóricos más generales como los de [15], [16] o [17]. El esquema de desarrollo de un modelo teórico se basa en estimar primero la radiación absorbida teniendo en cuenta las pérdidas ópticas que se producen en el trayecto de dicha radiación desde el plano de apertura hasta la superficie del tubo absorbedor. Posteriormente, las pérdidas térmicas suelen contabilizarse en términos de un coeficiente de pérdidas [18]. En el caso de que exista concentración, debe tenerse en cuenta que el mecanismo dominante para las pérdidas energéticas es el de radiación, regido por la ley de Steffan-Boltzmann, que se caracteriza por una dependencia de la cuarta potencia de la temperatura del colector. Además, debido a que los captadores suelen contar con una dimensión longitudinal de tamaño considerable, no se puede despreciar la variación que experimentan las propiedades físicas del fluido caloportador y los componentes del colector solar a lo largo de éste, debido a la variación de temperatura que el fluido va experimentando según circula por el tubo absorbedor. Estas son algunas de las razones que dificultan la obtención de una expresión para el rendimiento integral del conjunto.

El modelo de partida desarrollado en [1] tiene un carácter general que permite que sea aplicado a receptores térmicos de radiación solar de cualquier tecnología, tanto para concentradores cilindroparabólicos como para concentradores lineales Fresnel o receptores de torre central. En este trabajo nos centramos en los aspectos relativos a los CCP y a continuación revisaremos las características del modelo para este tipo concreto de receptores.

Para el desarrollo del modelo, se parte de un receptor consiste en un tubo desnudo de

diámetro D_{ro} (m) y longitud L (m). Una de las claves del modelo está en encontrar unos parámetros equivalentes, ε_{ext} y h_{ext} , denominados *emisividad equivalente de la superficie exterior del tubo* y *coeficiente de transferencia de calor convectivo equivalente* respectivamente, que permiten equiparar el comportamiento de un HCE real, con cubierta de vidrio, al modelo de tubo desnudo. Estos coeficientes deben hallarse previamente a partir de datos de laboratorio. A lo largo del desarrollo del modelo se realizan ciertas aproximaciones para las que se aportan justificaciones que no repetiremos ahora, pero que pueden encontrarse en el texto original. Como primera aproximación se considera que el receptor absorbe radiación de manera uniforme a través de toda su superficie. Igualmente se desprecia la transmisión de calor en la dirección axial.

El esquema propuesto parte del balance energético del modelo de tubo desnudo,
ec.(2.1)

$$\dot{q}_{perd}''(x) = \dot{q}_{abs}'' - \dot{q}_u''(x) \quad (2.1)$$

donde \dot{q}_{perd}'' es la energía perdida para una sección a una distancia x de la entrada al receptor, \dot{q}_{abs}'' es la radiación absorbida y \dot{q}_u'' es la energía útil . En la Fig.2,1 vemos una representación del modelo donde el sentido de flujo de la energía viene indicado por las flechas.

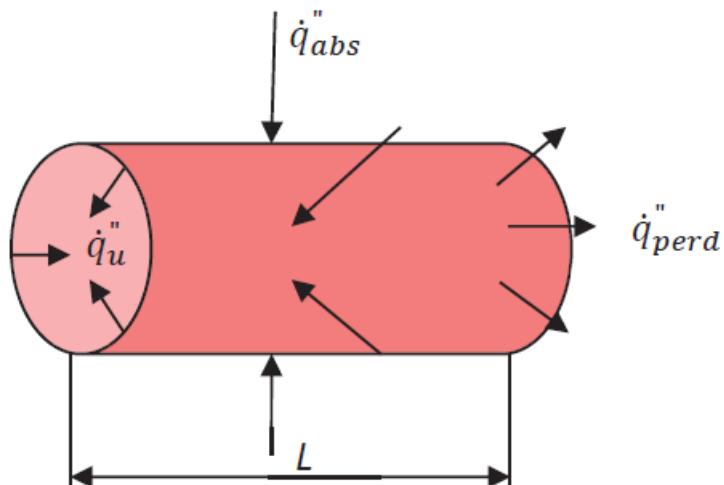


Fig. 2.1. Esquema del receptor empleado para el modelo. Fuente:[1]

La expresión para el cálculo de la radiación absorbida se muestra en la ec.(2.2).

$$\dot{q}_{abs}'' = \eta_{opt}(\theta) \cdot Cg \cdot DNI \cdot \eta_{sombras} \cdot \eta_{bordes} \quad (2.2)$$

El rendimiento óptico η_{opt} , las pérdidas geométricas η_{bordes} y las pérdidas por sombras, $\eta_{sombras}$ son valores conocidos, normalmente ofrecidos por los fabricantes, o que deben calcularse para cada momento en función de la geometría del receptor y la disposición de los concentradores en el campo solar. Cg es el factor de concentración y también es conocido a partir de la geometría del conjunto concentrador-receptor. Finalmente, DNI es la radiación normal incidente. La ec.(2.3) permite hallar al calor transferido desde el tubo absorbedor a temperatura T_{ro} , al fluido térmico a temperatura T_f :

$$\dot{q}_u''(x) = U_{rec} \cdot [T_{ro}(x) - T_f(x)] \quad (2.3)$$

donde U_{rec} es el coeficiente global de transferencia de calor hacia el interior, cuya expresión se muestra en la ec.(2.4):

$$U_{rec} = \frac{1}{\frac{1}{h_{int}} + \frac{D_{ro} \cdot \ln(\frac{D_{ro}}{D_{ri}})}{2 \cdot k_{rec}}} \quad (2.4)$$

Como aproximación se considera que U_{rec} ($W/m^2 \cdot K$) es constante a lo largo de la longitud del tubo. D_{ro} y D_{ri} son el diámetro exterior e interior respectivamente del tubo absorbedor, (m). h_{int} ($W/m^2 \cdot K$) es el coeficiente de transferencia de calor convectivo hacia el interior y k_{rec} es conductividad del material del receptor, en ($W/m \cdot K$)

Para el cálculo de las pérdidas de calor se empleará la ec.(2.5), en la que se tiene en cuenta un término radiativo, con dependencia de la cuarta potencia de las temperaturas, y otro convectivo con dependencia de 1^{er} grado.

$$\dot{q}_{perd}''(x) = \sigma \cdot \varepsilon_{ext} \cdot (T_{ro}^4(x) - T_{ext}^4) + h_{ext} \cdot (T_{ro}(x) - T_{ext}) \quad (2.5)$$

donde σ es la constante de Stefan-Boltzmann ($5,67 \times 10^{-8} W/m^2 K^4$), ε_{ext} es la emisividad del tratamiento superficial exterior y h_{ext} es el coeficiente de convección exterior. Estas dos últimas constantes son características de cada receptor y variables, por ejemplo,

en función de las condiciones de degradación del recubrimiento selectivo o del viento exterior. Deben ser halladas experimentalmente en laboratorio.

La última ecuación necesaria es la ec.(2.6), donde se calcula el incremento de temperatura que experimenta el fluido, considerando despreciables los cambios en energía cinética y un calor específico a presión constante, c_p (J/kg·K). Denominando $T_f(x)$ a la temperatura del fluido en la sección a distancia x de la entrada y T_{in} a la temperatura del fluido a la entrada, tenemos:

$$\pi \cdot D_{ro} \cdot x \cdot \dot{q}_{abs}'' \cdot \eta(x) = \dot{m} \cdot c_p \cdot (T_f(x) - T_{in}) \quad (2.6)$$

Con esta última ecuación se calcula el rendimiento integral hasta una sección a una distancia x de la entrada, $\eta(x)$. Finalmente, a partir del rendimiento local, dado por la ec.(2.7) podemos calcular el rendimiento integral mediante la ec.(2.8).

$$\eta_x(x) = \frac{\dot{q}_u''(x)}{\dot{q}_{abs}''} \quad (2.7)$$

$$\eta(x) = \frac{\int_0^x \eta_x(x) dx}{\int_0^x dx} \quad (2.8)$$

donde desarrollando $\eta_x(x)$ según la ec.(2.9):

$$\eta_x(x) = \eta(x) + \eta'(x) \cdot x \quad (2.9)$$

y normalizando la distancia a la unidad con la variable adimensional $x^* = x/L$, obtenemos la ecuación integral ec.(2.10):

$$\eta(x^*) = 1 - \frac{\int_0^{x^*} \dot{q}_{perd}''(dx^*) \cdot dx^*}{\dot{q}_{abs}'' \cdot dx^*} \quad (2.10)$$

La resolución de esta ecuación requiere un largo desarrollo, en el que se introducen nuevos factores característicos del sistema, que puede encontrarse en la obra de referencia, por lo que la omitiremos aquí. Pese a la complejidad de la expresión final obtenida, que dificulta extraer conclusiones de manera directa, el modelo incorpora todos los parámetros característicos del sistema y lo hace manteniendo su sentido físico. A partir de la solución

se puede obtener una expresión para el modelo local (rendimiento en una sección determinada del absorbedor) y una expresión para el modelo de colector completo, es decir, un rendimiento integral a lo largo de todo el absorbedor. Esta última expresión es la que nos interesa. A continuación se presenta la ecuación del Modelo de 4º Orden completo del colector en la ec.(2.11) y sucesivamente las ecuaciones que definen sus parámetros:

$$\eta(x^*) = \frac{\eta_0 \cdot g'(Z)}{1 - g'(Z)} \cdot \frac{1}{NTU \cdot x^*} \cdot \left(e^{\frac{1-g'(Z)}{g'(Z)} \cdot NTU \cdot x^*} - 1 \right) - \frac{\eta_0^2}{6} \cdot \frac{g''(Z)}{g'(Z)} \cdot NTU^2 \cdot x^{*2} - \frac{\eta_0^3}{24} \cdot \frac{g'''(Z)}{g'(Z)} \cdot NTU^3 \cdot x^{*3} \quad (2.11)$$

$$\eta_0 = 1 - (f_1 \cdot Z + f_2 \cdot Z^2 + f_3 \cdot Z^3 + f_4 \cdot Z^4) \quad (2.12)$$

$$Z = \eta_0 + \frac{1}{f_0} \quad (2.13)$$

$$f_0 = \frac{\dot{q}_{abs}''}{U_{rec} \cdot (T_{in} - T_{ext})} \quad (2.14)$$

$$g(Z) = - \left(1 + \frac{1}{f_0} \right) + (1 + f_1) \cdot Z + f_2 \cdot Z^2 + f_3 \cdot Z^3 + f_4 \cdot Z^4 \quad (2.15)$$

$$g'(Z) = 1 + f_1 + 2 \cdot f_2 \cdot Z + 3 \cdot f_3 \cdot Z^2 + 4 \cdot f_4 \cdot Z^3 \quad (2.16)$$

$$g''(Z) = 2 \cdot f_2 \cdot Z + 6 \cdot f_3 \cdot Z + 12 \cdot f_4 \cdot Z^2 \quad (2.17)$$

$$g'''(Z) = 6 \cdot f_3 + 24 \cdot f_4 \cdot Z \quad (2.18)$$

$$g^{IV}(Z) = 24 \cdot f_4 \quad (2.19)$$

$$f_1 = \frac{4 \cdot \sigma \cdot \varepsilon_{ext} \cdot T_{ext}^3 + h_{ext}}{U_{rec}} \quad (2.20)$$

$$f_2 = 6 \cdot T_{ext}^2 \cdot \left(\frac{\sigma \cdot \varepsilon_{ext}}{U_{rec}} \right) \cdot \left(\frac{\dot{q}''_{abs}}{U_{rec}} \right) \quad (2.21)$$

$$f_3 = 4 \cdot T_{ext} \cdot \left(\frac{\sigma \cdot \varepsilon_{ext}}{U_{rec}} \right) \cdot \left(\frac{\dot{q}''_{abs}}{U_{rec}} \right)^2 \quad (2.22)$$

$$f_4 = \left(\frac{\sigma \cdot \varepsilon_{ext}}{U_{rec}} \right) \cdot \left(\frac{\dot{q}''_{abs}}{U_{rec}} \right) \quad (2.23)$$

Para la resolución de este modelo es preciso conocer previamente diferentes parámetros, muchos de los cuales pueden obtenerse directamente de las características geométricas y físicas de los materiales con los que está construido el HCE. De especial importancia son ε_{ext} y h_{ext} pues son dos coeficientes que de forma global vienen a caracterizar las pérdidas energéticas del receptor. Para obtener las ecuaciones que los caracterizan se parte de la expresión del coeficiente global del pérdidas al exterior dada por la ec.(2.24) de la siguiente forma:

$$U_{ext} = h_{ext} + \sigma \cdot \varepsilon_{ext} \cdot (T_{ro}^2 + T_{ext}^2) \cdot (T_{ro} + T_{ext}) \quad (2.24)$$

Tal y como se ha indicado al comienzo de este capítulo, es necesario realizar ensayos de laboratorio bajo diferentes condiciones de viento (W_{spd}) y temperatura exterior (T_{ext}) para obtener el flujo de calor de pérdidas y calcular así dos expresiones del tipo $\varepsilon_{ext}(T_{ext}, W_{spd})$ y $h_{ext}(T_{ext}, W_{spd})$. Para este trabajo se emplean los valores obtenidos en [1] a partir de [13], [14] y [19].

A partir de este modelo de 4º Orden se realiza un desarrollo que permite obtener dos modelos simplificados de colector completo: el Modelo de Primer Orden y el Modelo Simplificado.

2.2. Modelo de Primer Orden

La ec.(2.25) presenta el Modelo de Primer Orden. Para llegar a ella resuelve la ec.(2.12) despreciando monomios a partir de segundo grado, con lo que se puede sustituir el rendi-

miento a la entrada del absorbedor, η_0 , por su valor aproximado dado en la ec.(2.26):

$$\eta(x^*) = \left[1 - \frac{\dot{q}_{crit}''}{\dot{q}_{abs}''} \right] \cdot \frac{1}{NTU_{perd} \cdot x^*} \cdot \left(1 - e^{-NTU_{perd} \cdot F'_{crit} \cdot x^*} \right) \quad (2.25)$$

$$\eta_0 = F'_{crit} \cdot \left[1 - \frac{\dot{q}_{crit}''}{\dot{q}_{abs}''} \right] \quad (2.26)$$

F'_{crit} se asemeja al parámetro empleado en el modelo desarrollado por Hottel y Whillier en [15].

$$F'_{crit} = \frac{1}{\frac{4 \cdot \sigma \cdot \varepsilon_{ext} \cdot T_{in}^3}{U_{rec}} + \frac{h_{ext}}{U_{rec}} + 1} = \frac{1}{\frac{U_{crit}}{U_{rec}} + 1} \quad (2.27)$$

Y NTU_{perd} se obtiene mediante la ec.(2.28):

$$NTU_{perd} = \frac{(4 \cdot \sigma \cdot \varepsilon_{ext} \cdot T_{in}^3 + h_{ext}) \cdot A_{ext}}{\dot{m} \cdot c_p} = \frac{U_{crit} \cdot A_{ext}}{\dot{m} \cdot c_p} \quad (2.28)$$

Además, en (2.26) y (2.27) se han reagrupado variables en diferentes términos que cuentan con sentido físico. De este modo, se definen:

$$\dot{q}_{crit}'' = \sigma \cdot \varepsilon_{ext} \cdot (T_{in}^4 - T_{ext}^4) + h_{ext} \cdot (T_{in} - T_{ext}) \quad (2.29)$$

y

$$U_{crit} = 4 \cdot \sigma \cdot \varepsilon_{ext} \cdot T_{in}^3 + h_{ext} \quad (2.30)$$

Los coeficientes \dot{q}_{crit}'' y U_{crit} son valores de referencia en el estudio del comportamiento del colector, pues cuando \dot{q}_{abs}'' se aproxima a \dot{q}_{crit}'' el rendimiento del colector se hace nulo y cuando U_{rec} se aproxima a U_{crit} , el rendimiento aproxima a la mitad, tal y como se aprecia si sustituimos (2.27) en (2.26) para obtener (2.31):

$$\eta_0 = \frac{1}{\frac{U_{crit}}{U_{rec}} + 1} \cdot \left[1 - \frac{\dot{q}_{crit}''}{\dot{q}_{abs}''} \right] \quad (2.31)$$

El Modelo de Primer Orden presenta la ventaja de que el cálculo de $\eta(x^*)$ es explícito, con la reducción del coste computacional que esto conlleva. Por otro lado, también nos ofrece una forma de calcular un valor aproximado del rendimiento a la entrada del colector, η_0 .

2.3. Modelo simplificado

Si se desarrolla por Taylor la función exponencial del Modelo de Primer Orden, se trunca por el segundo término y se sustituye \dot{q}_{abs}'' por su expresión en función de *DNI* se obtiene la ec.(2.32) para el cálculo del rendimiento para la totalidad del receptor, η_T , mediante el Modelo Simplificado:

$$\begin{aligned}\eta_T &= F'_{crit} \cdot \left[1 - \frac{\dot{q}_{crit}''}{\dot{q}_{abs}''} \right] \\ &= \frac{F'_{crit}}{Cg} \cdot \left[Cg \cdot IAM \cdot \cos(\theta) \cdot \eta_{opt,pico} \cdot \eta_{sombras} \cdot \eta_{bordes} - \frac{h_{ext} \cdot (\bar{T}_f - T_{ext})}{DNI} \right. \\ &\quad \left. - \frac{\sigma \cdot \varepsilon_{ext} \cdot (\bar{T}_f^4 - T_{ext}^4)}{DNI} \right]\end{aligned}\quad (2.32)$$

Esta ecuación es más parecida a la encontrada en otros modelos de diferentes autores, como por ejemplo en [15] o [16], pero con dependencia de la cuarta potencia de la temperatura media del fluido, \bar{T}_f , lo cual tiene mayor sentido físico al esperarse que las pérdidas radiativas sean dominantes en situaciones de media y alta concentración.

Veremos más adelante que el Modelo Simplificado es de aplicación más restringida y los resultados que se obtienen con él comienzan a desviarse de modelos más exactos, como el de 4º Orden, cuando aumenta el tamaño de malla de integración, la temperatura de trabajo o la relación de concentración.

2.4. Aplicabilidad de los modelos

Aunque en el caso del Modelo de 4º Orden no se ha realizado ninguna simplificación para la resolución de la ecuación característica, sí que se han hecho las siguientes consideraciones que limitan su aplicación:

- Se ha considerado que los parámetros característicos U_{rec} , ε_{ext} , h_{ext} y Cp son constantes a lo largo de toda la longitud del receptor. Se considerará que esto es aceptable para longitudes inferiores a 100 m tal y como se indica en el desarrollo del modelo.

- Se supone uniformidad del flujo de radiación sobre el tubo absorbedor. Para tecnología CCP se acepta esta hipótesis.
- La caracterización de los tubos absorbedores empleados en CCP es compatible con el desarrollo del modelo basada en un tubo desnudo (para los que posteriormente se emplearán unos coeficientes de transmisión de calor adecuados para los tubos con cubierta de vidrio).
- La suposición de fluido incompresible, en la que se desprecia el término de pérdida de carga y de energía cinética sobre el término energético, es adecuada para plantas que operan con aceite térmico, dado que los circuitos están presurizados para mantener en todo momento el fluido en estado de líquido saturado y los caudales de operación tienen un número de Reynolds alto que garantiza un estado de mezcla homogénea.
- El flujo puede suponerse uniforme en el interior del tubo absorbedor.
- El reducido gradiente de temperatura y espesor del tubo absorbedor permiten que pueda despreciarse el efecto de transmisión de calor longitudinal.

Según lo visto, el modelo resulta aplicable a la simulación de un campo solar de concentradores cilindroparabólicos bajo las condiciones normales de operación. Por otro lado, la simulación se realizará también para el cálculo con intervalos horarios en los que se supondrá condiciones estacionarias de planta y se descartarán aquellos períodos de arranque y parada o cambios abruptos en los que las inercias propias del sistema y la intervención de los operadores de planta producirían que el comportamiento instantáneo no se correspondiese con el simulado.

El Modelo de Primer Orden presenta resultados algo menos precisos que el Modelo de 4º Orden, en general.

El Modelo Simplificado solo es válido además para longitudes de integración más reducidas.

3. MODELADO DEL CAMPO SOLAR

3.1. Metodología seguida para el modelado del campo solar

El software desarrollado se basa en el paradigma de Programación Orientada a Objetos (POO) donde, a grandes rasgos, cada sistema físico se define como un objeto perteneciente a una Clase con la capacidad de recibir información, manipularla de acuerdo a unas reglas propias del sistema y devolver información.

Una de las principales ventajas de esta metodología es la modularidad, de tal forma que se puede ir desarrollando jerárquica, progresiva e independientemente cada uno de los sistemas para después interconectarlos. Posteriormente se puede modificar el comportamiento de alguno de estos objetos re-programando la Clase a la que pertenece sin que esto afecte de forma drástica al resto de objetos del modelo. Es una técnica escalable y que permite definir diferentes grados de intervención al usuario final, desde interactuar con cada objeto como si de una caja negra se tratase hasta modificar el comportamiento del sistema introduciendo sus propios métodos en las clases.

Por todas estas razones, el modelado mediante POO resulta muy interesante para la simulación de sistemas en el ámbito de la ingeniería y ha sido el elegido para el desarrollo del código de este TFG

3.2. Sistemas físicos y Clases para el modelado del campo solar

En los siguientes apartados iremos describiendo el campo solar desde el punto de vista de su comportamiento físico, los subsistemas que lo componen y definiremos las Clases que se deben programar para modelar cada uno de estos subsistemas. Pero en primer lugar aclararemos alguna terminología en el contexto sistema-modelo.

Se denomina HCE a cada uno de los tubos absorbedores de unos 4 m de longitud con envolvente de vidrio propia que, soldados uno tras otro, forman la tubería sobre la que se concentra la radiación solar. Los HCE se montan sobre unidades estructurales denominadas SCE (Solar Collector Element). El modelo del HCE puede tener una dimensión longi-

tudinal más flexible, de tal forma que una instancia de la clase HCE tenga una longitud de uno o varios HCE. Esto nos permitirá jugar con el tamaño de la malla de integración para el cálculo del rendimiento integral de un concentrador completo (formado, en realidad, por un conjunto de HCEs).

Un conjunto de SCEs que se mueven solidariamente entre ellos pero con capacidad de movimiento independiente de otro conjunto de SCEs se denomina SCA (Solar Collector Assembly). El tubo absorbedor montado en cada SCA está unido mediante uniones móviles al tubo absorbedor del siguiente SCA o a las tuberías de entrada y salida del lazo. El SCA es, por tanto, la unidad mínima de seguimiento solar.

Un conjunto de SCAs con su tubo absorbedor conectado en serie constituye un lazo (Clase Loop). Cada lazo consta de un número suficiente de SCAs para garantizar que, bajo condiciones de diseño, el fluido caloportador alcanza la temperatura deseada a la salida del lazo, es decir, se produce el salto térmico necesario demandado por el proceso consumidor de calor o el ciclo termodinámico de generación de energía eléctrica. Si la temperatura en el SCA sobrepasa la máxima permitida, el SCA puede desenfocar parcial o totalmente con el fin de dejar de concentrar radiación sobre el tubo absorbedor, limitando el aporte de calor y estabilizando la temperatura de salida del fluido.

Las Clases que se describen a continuación se recogen en un fichero que puede funcionar a modo de librería. Esta librería puede referenciarse para la creación de instancias de cada tipo de objeto. Se ha denominado *csenergy* a esta librería.

En la Fig.3.1 se muestra esquemáticamente la relación entre las clases HCE, SCA y Loop. La clase Loop mantiene una referencia al conjunto ordenado (lista) de SCA que lo forman y, a su vez, la clase SCA mantiene una referencia a la lista de HCEs que la forman. En sentido ascendente, cada HCE mantiene una referencia al SCA del que forma parte y cada SCA una referencia a su lazo.

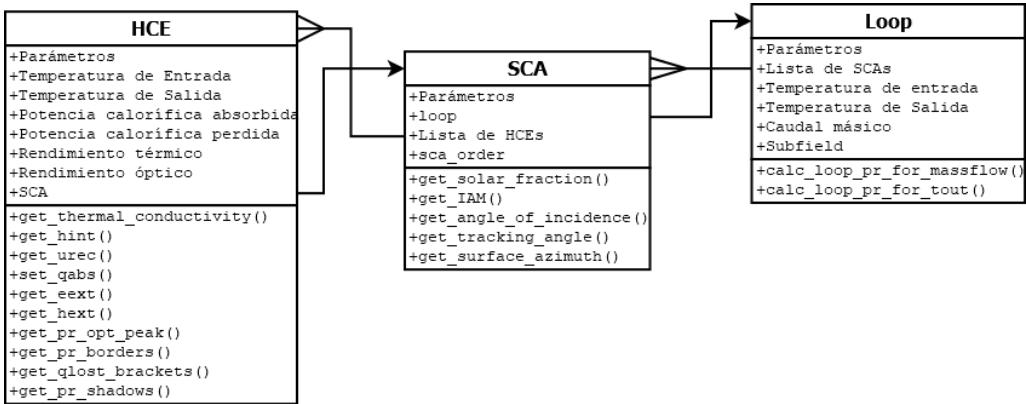


Fig. 3.1. Esquema relacional de las Clases HCE, SCA y Loop. Se muestran los principales atributos y métodos de cada clase

Un subcampo o sección es un conjunto de lazos conectados en paralelo, de tal forma que se espera que el caudal que circula por cada uno de sus lazos sea el mismo. El subcampo cuenta con válvulas de regulación de caudal a su entrada, por lo que constituye la unidad mínima de control de caudal en el campo solar. En algunas ocasiones cada lazo tiene capacidad de regulación de su caudal de forma constante. En ese caso se podría decir que cada lazo actúa como un subcampo con un único lazo, pero esto no es lo habitual.

Finalmente, el campo solar está formado por un conjunto de subcampos. El fluido caloportador frío entra en el campo solar y se distribuye por cada uno de los subcampos, donde se vuelve a distribuir equitativamente entre los lazos. En los lazos, el HTF se calienta y retorna a una tubería que lo conduce a la salida del subcampo, donde finalmente el HTF procedente de todos los subcampos se mezcla y se transporta, a lo largo de una tubería denominada colector caliente, hasta el punto de consumo. En la Fig.3.2 se esquematiza esta estructura de agregación de elementos. Cada Clase cuenta con métodos que permiten calcular los valores agregados de caudal, potencia y temperatura del fluido a partir de las aportaciones de cada uno de los subsistemas que engloba.

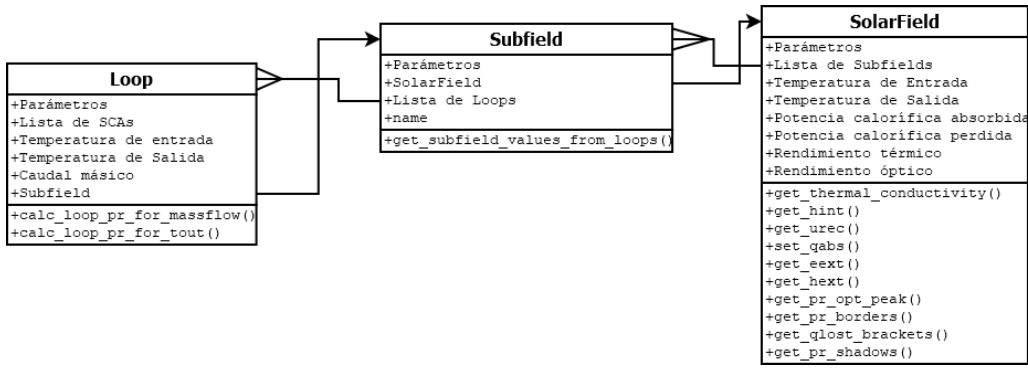


Fig. 3.2. Esquema relacional de las Clases SolarField, Subfield y Loop. Se muestran los principales atributos y métodos de cada clase

3.2.1. Clases derivadas de Model: Clase **ModelBarbero4thGrade**, **ModelBarbero1stGrade** y **ModelBarberoSimplified**

Se emplean clases derivadas de la Clase Model para implementar los diferentes modelos empleados para calcular el rendimiento y para simular, por tanto, el funcionamiento de cada HCE. Es en cada una de estas clases donde se desarrolla el algoritmo que, a partir de los parámetros físicos que definen al HCE, las variables que definen el estado del HTF que circula por él y las condiciones de operación, resuelve las ecuaciones definidas en el modelo y nos permite conocer el rendimiento HCE.

Clase **ModelBarbero4thOrder**

La instancia de esta clase recibe como valores de entrada una referencia a una instancia de un HCE del cual va a calcular su rendimiento, una referencia a la instancia del HTF que se está empleando y valores de condiciones meteorológicas de radiación, temperatura y velocidad del viento. El HCE debe estar inicializado previamente con los valores de caudal másico, temperatura y presión de entrada y el flujo de calor absorbido \dot{q}_{abs} .

El procedimiento de cálculo implementado en el método *calc_pr()* es el siguiente (los parámetros que se obtienen mediante métodos propios de las instancias del HCE y del HTF se explican en los apartados correspondiente más adelante):

- Estimación de la temperatura de pared exterior del tubo absorbedor T_{ro} según la ec(3.1) a partir del coeficiente de transmisión de calor al interior U_{rec} y del flujo de

radiación absorbido por el tubo absorbedor, \dot{q}_{abs}'' , a partir de la instancia del HCE. Para el primer HCE del lazo se asume un rendimiento inicial $\eta = 1$ pero para los siguientes se emplea el rendimiento del HCE anterior, con lo cual se acelera un poco el proceso de convergencia por partirse de un valor previsiblemente más próximo.

$$T_{ro} = T_f + \eta \cdot \frac{\dot{q}_{abs}''}{U_{rec}} \quad (3.1)$$

- Cálculo del flujo de pérdidas \dot{q}_{perd}'' mediante la ec.(2.5) incrementado con las pérdidas a través de los soportes que sujetan el tubo absorbedor, $\dot{q}_{perd,soportes}$. Las pérdidas en los soportes se modelan mediante la ec.(3.13) que se explica en el apartado correspondiente de la clase HCE.
- Cálculo de los parámetros de funcionamiento \dot{q}_{crit}'' , U_{crit} y NTU para el HCE con la temperatura de pared calculada previamente según las ecuaciones (2.29) y (2.30) respectivamente.
- Cálculo de los coeficientes f_1 , f_2 , f_3 y f_4 mediante las ecuaciones (2.20) a (2.23) y cálculo de f_0 mediante la ec.(2.14).
- Se resuelve la ec.(2.12) con de forma iterativa mediante Newton-Raphson para calcular η_0 . Como valor inicial se calcula η_0 a partir de la ec.(2.26) del Modelo de 1^{er} Orden.
- Con el valor de η_0 obtenido se calculan los valores de Z , $g'(Z)$, $g''(Z)$ y $g'''(Z)$ dados por las ecuaciones (2.13) a (2.18).
- Finalmente, se calcula el rendimiento $\eta(x^*)$ según la ec.(2.11), la temperatura de pared exterior T_{ro} y se comparan con los valores iniciales. Si las diferencias son superiores a cierto margen configurable se vuelve a realizar otra iteración hasta conseguir la convergencia, pero previamente a cada iteración se re-calculan todos los pasos anteriores empleando la temperatura de pared del tubo absorbedor calculada con el nuevo rendimiento.

Al inicio del proceso iterativo se considera que, para el primer HCE del lazo, la temperatura del fluido T_f es igual a la temperatura de entrada al lazo (y al HCE, por tratarse del primero). Para los siguientes HCEs del lazo se parte de una temperatura del fluido igual

a la temperatura de salida del HCE anterior pero incrementada con la mitad del salto de temperatura que experimentó dicho HCE.

Una vez finalizado el proceso iterativo, la instancia del HCE actualiza sus valores de rendimiento, temperatura y presión de salida del HTF, quedando totalmente definido su punto de funcionamiento. Las condiciones de temperatura y presión a la salida del HCE serán las de entrada del HCE siguiente.

Al calcular el rendimiento integral para todo la longitud del HCE estamos haciendo coincidir el tamaño de la malla de integración con la longitud física real del HCE. Se ha comprobado que la reducción de la malla no aumenta de forma apreciable la precisión de los cálculos y en cambio sí supone un coste computacional importante. Por el contrario, una forma de acelerar el proceso de simulación consiste en considerar artificialmente que la longitud del HCE es mayor que la real. Se trata de aumentar el tamaño de la malla de integración para reducir el número de cálculos. En este trabajo se seguirá, al igual que en [1], el criterio de no superar un tramo de HCE superior a 100 m propuesto en [20] para análisis unidimensional.

Clases ModelBarbero1stOrder y ModelBarberoSimplified

El proceso para el cálculo del rendimiento que realizan estas dos clases es similar al del Modelo de 4º Orden, salvo que ahora, el cálculo del rendimiento es explícito y no es necesario recurrir a la resolución de las ecuaciones por métodos numéricos, sino que se emplean directamente las ecuaciones (2.25) y (2.32) respectivamente. Una vez calculado el rendimiento, el algoritmo recalcula los valores hallados al inicio de la iteración y si la diferencia es mayor a los valores consignados, realiza una nueva iteración hasta alcanzar la convergencia deseada.

3.2.2. Clase HCE para modelado del Heat Collector Element

De cara a modelar el funcionamiento del HCE como elemento responsable de calentar el HTF de forma compatible con el Modelo físico desarrollado, se define la Clase HCE, entre cuyos atributos se encuentran: temperatura de entrada del HTF, T_{in} (internamente el programa trabaja con K como unidades de temperatura, aunque la entrada y salida de

datos se realiza con $^{\circ}C$); potencia calorífica absorbida a lo largo de todo el HCE, \dot{q}_{abs} (W); potencia calorífica perdida a lo largo del HCE, \dot{q}_{perd} (W); rendimiento óptico del conjunto HCE+SCA, η_{opt} ; rendimiento térmico, η y temperatura de salida, T_{out} , (K).

El caudal másico de HTF que circula por el HCE, \dot{m} (kg/s) se obtiene mediante una referencia al SCA y, seguidamente, otra referencia al Loop que contienen al HCE.

Con estos parámetros el comportamiento del HCE queda totalmente caracterizado en el sistema desde el punto de vista del proceso de generación. Estos atributos (pueden entenderse como variables) están relacionados entre sí según las reglas que aplique cada modelo. La temperatura de salida del HTF, T_{out} , aparece implícita en la ec.3.2:

$$\dot{H} = \dot{m} \int_{T_{in}}^{T_{out}} c_p(T) dT \quad (3.2)$$

donde \dot{H} representa una tasa temporal de incremento de entalpía, pues hemos considerado que se trata de un fluido incompresible y también se ha despreciado la participación de energía cinética, por lo que la potencia térmica se invierte en incrementar la entalpía del sistema. Previamente se debe calcular \dot{H} según la ec.(3.3):

$$\dot{H} = \dot{q}_{abs}'' \cdot \eta \cdot \pi \cdot D_{ro} \cdot L \cdot \gamma_L \cdot \gamma_g \quad (3.3)$$

La ec.(3.2) puede resolverse por métodos numéricos si el calor específico C_p del fluido se ha obtenido a partir de un polinomio. En el caso de que se disponga de una función que proporcione la temperatura del fluido en función de la entalpía $T(h)$, como ocurre si se usa *CoolProp*, se puede calcular su valor directamente a través de las funciones que ofrece esta librería como $T_{out} = T(h_{out})$.

En la ec.3.3 se introducen el *factor de longitud efectiva*, γ_L y *factor de interceptación geométrico*, γ_g para tener en cuenta la reducción de la longitud *activa* del HCE debido a los fuelles en los extremos del HCE y al sombreado del escudo térmico en las uniones de HCEs. Un valor típico para ambos factores está comprendido entre 0,96 y 0,97 [21]. En el caso de que el calor absorbido sea nulo, la temperatura de salida será inferior a la de entrada y el valor $\dot{H} < 0$. En este caso, no existe reducción de la longitud efectiva del absorbedor y la energía perdida se calcula según la ec.(3.4) pues a lo largo de toda la

superficie del HCE se experimentan pérdidas energéticas:

$$\dot{H} = \dot{q}_{perd}'' \cdot \pi \cdot D_{ro} \cdot L \quad (3.4)$$

Cálculo del flujo de calor absorbido, \dot{q}_{abs}''

La radiación que alcanza al fluido caloportador puede obtenerse mediante la ec.2.2:

$$\dot{q}_{abs}'' = \eta_{opt}(\theta) \cdot Cg \cdot DNI \cdot \eta_{sombras} \cdot \eta_{bordes} \quad (2.2)$$

donde DNI es la radiación normal directa cuyo valor se lee para cada fecha de cálculo de la simulación. Cg es el factor de concentración geométrica, definido genéricamente para sistemas de concentración como el cociente entre el área de apertura del concentrador, A_c y la superficie del receptor, A_{ext} . Hemos considerado como efectiva toda el área del receptor, no solo aquella donde se concentra la radiación, ya que supondremos que el flujo se reparte uniformemente por toda la superficie de tubo absorbedor. De esta manera, el área de apertura de un concentrador de longitud L , con una longitud de apertura de su superficie parabólica A_p , viene dado por la ecuación:

$$A_c = A_p \cdot L \quad (3.5)$$

y la superficie del receptor, de igual longitud L y diámetro exterior del tubo absorbedor D_{ro} , es:

$$A_{ext} = \pi \cdot D_{ro} \cdot L \quad (3.6)$$

De esta manera, el factor de concentración geométrica para un colector cilindroparabólico se calcula según la ec.(3.7):

$$Cg = \frac{A_p}{\pi \cdot D_{ro}} \quad (3.7)$$

El rendimiento óptico $\eta_{opt}(\theta)$ depende del ángulo de incidencia θ y se obtiene a partir del rendimiento óptico pico, $\eta_{opt,peak}$ y del modificador del ángulo de incidencia, IAM

según la ec.(3.8):

$$\eta_{opt}(\theta) = \eta_{opt,peak} \cdot IAM \cdot \cos(\theta) \quad (3.8)$$

Esta ecuación incluye también el coseno del ángulo de incidencia pues consideraremos que, en general, no está incluido este término dentro de IAM. En caso de que la expresión del *IAM* ofrecida por el fabricante ya incluyese este efecto, debería eliminarse de la ecuación (3.8). Para calcular $\eta_{opt,peak}$ empleamos la expresión dada en la ec.(3.9).

$$\eta_{opt,peak} = \alpha \cdot \tau \cdot \rho \cdot \gamma \quad (3.9)$$

La ecuación para *IAM* se ofrece en la sección correspondiente al modelado del SCA ya que es una propiedad más propia del sistema de concentración y seguimiento. La instancia del HCE hace una llamada al método *get_IAM* de su SCA asociado, aquel en el que está montado, para recibir su valor. Igualmente, en la ec.(3.9) los parámetros ρ (reflectividad del concentrador) y γ (fracción solar), son parámetros del SCA y deben obtenerse de la instancia de SCA asociada al HCE. α es la absorvidad del receptor y τ es la transmisividad del vidrio envolvente del tubo absorbedor. En ambos casos se trata de parámetros configurables que se introducen con el resto de características del HCE en el archivo de configuración de la simulación.

El factor η_{bordes} contabiliza las pérdidas debidas a que en una pequeña porción del tubo absorbedor del SCA no se produce concentración debido al ángulo de incidencia. Un tramo del tubo absorbedor, que puede implicar desde solo un tramo del primer HCE hasta a varios HCEs, tendrá un flujo de radiación nulo, o muy bajo. El tramo de tubo absorbedor que queda sin concentración (L_{bordes} se calcula mediante la ec.(3.10) a partir de la distancia focal, f_l y de ángulo de incidencia θ :

$$L_{bordes} = \frac{f_l}{\tan(\theta)} \quad (3.10)$$

A partir de este valor el código calcula que fracción del HCE o cuantos HCEs quedan inutilizados y les asigna un rendimiento nulo.

En el caso del factor $\eta_{sombras}$, se trata de un valor que se calcula en base a la porción del concentrador que se encuentra afectado por sombras debido a que la distancia de sepa-

ración entre lazos está acotada. En disposiciones de lazos habituales con eje seguimiento Norte-Sur estas sombras solo aparecen a primera y última hora del día. Su cálculo exacto según se describe en [22] requeriría conocer completamente la disposición de cada lazo en cada instante, pero una aproximación suficiente se puede conseguir según la ec.(3.11) tal y como hace SAM, [23]:

$$\eta_{sombras} = \frac{\cos(\beta) \cdot D_L}{A_c} \quad (3.11)$$

donde β es el ángulo de seguimiento, D_L es la distancia de separación entre lazos y A_c es la apertura del concentrador.

Pérdidas en el tubo absorbedor, \dot{q}_{perd}''

Las pérdidas se modelan según la ec.(2.5), que repetimos a continuación:

$$\dot{q}_{perd}''(x) = \sigma \cdot \varepsilon_{ext} \cdot (T_{ro}^4(x) - T_{ext}^4) + h_{ext} \cdot (T_{ro}(x) - T_{ext}) \quad (2.5)$$

donde ε_{ext} es la *emisividad equivalente de la superficie exterior* del receptor. Depende de la temperatura de pared exterior del tubo y se emplea la ec.(3.12) para calcularla:

$$\varepsilon_{ext} = A_0 + A_1 \cdot (T_{ro} - 273,15) \quad (3.12)$$

Se corrige ligeramente su valor en función de la velocidad del viento, incrementando su valor un 1 % con un viento de 4 m/s y un 2 % para viento de 7 m/s. Los coeficientes A_0 y A_1 son los que se ofrecen en [1].

Por otro lado, h_{ext} , es el *coeficiente de transferencia de calor convectivo equivalente al exterior*. Su valor puede considerarse nulo para el caso de un HCE con vacío en su espacio anular. Nuevamente, en [1] se ofrecen las ecuaciones para diferentes combinaciones de recubrimiento, Black-Chrome o Cermet y conservación o no del vacío, obtenidas mediante simulación CFD (Computational Fluid Dynamics) por su autor para un modelo unidimensional del HCE. A falta de datos para los modelos concretos empleados en este trabajo, en todas nuestras simulaciones consideraremos que su valor es nulo (equivalente

a un caso de velocidad de viento nula), lo cual es aceptable en etapas de prediseño de plantas o durante análisis paramétrico.

Se incluye en el modelado el cálculo de las pérdidas a través de los soportes que sujetan al HCE, $\dot{q}_{perd,soportes}$. Su peso relativo en el total de pérdidas del campo no es muy elevado, por lo que, a falta de más datos, se hace uso de la ec.(3.13) propuesta en [20]:

$$\dot{q}_{perd,soportes} = n \cdot \frac{\sqrt{P_b \cdot k_b \cdot A_{cs,b} \cdot \bar{h}_b} \cdot (T_{base} - T_{ext})}{L} \quad (3.13)$$

donde P_b es el perímetro de la sección del soporte ($0,2032\text{ m}$), $A_{cs,b}$ es la sección transversal de la unión entre el brazo y el tubo absorbedor ($1,613 \cdot 10^{-4}\text{ m}^2$), K_b es la conductividad térmica del acero empleado en el brazo ($48,0\text{ W}/(\text{m}\cdot\text{K})$), \bar{h}_b es el coeficiente de transmisión de calor por convección medio hacia el exterior ($20\text{ W}/(\text{m}^2\cdot\text{K})$), T_{base} es la temperatura de la zona de conexión entre los brazos y el tubo absorbedor, L es la longitud del colector y n es el número de soportes por colector.

Otros atributos y métodos de la Clase HCE

Ya hemos visto, al hablar de la Clases para los modelos, cómo un objeto (instancia) de la clase HCE puede ser procesada por otra instancia de la clase del modelo para simular su comportamiento. Es necesario que la instancia del HCE pase los siguientes parámetros al modelo:

- k_{rec} , conductividad térmica de la pared del receptor. Se ha empleado la ec.(3.14) válida para el acero inoxidable 321H:

$$k_{rec} = 0,0153 \cdot (T - 273,15) + 14,77 \quad (3.14)$$

- h_{int} , coeficiente de transferencia de calor convectivo hacia el interior. Para el cálculo se emplea la ec.(3.15) donde Nu_G es el número de Nusselt obtenido mediante la correlación de Gnielinski dada en la ec.(3.16), D_{ri} es el diámetro interior del tubo absorbedor y k_f es la conductividad térmica a la temperatura del fluido:

$$h_{int} = \frac{Nu_G \cdot k_f}{D_{ri}} \quad (3.15)$$

$$Nu_G = \frac{\left(\frac{C_f}{2}\right) \cdot (Re_{D_{ri}} - 1000) \cdot Pr_f}{1 + 12,7 \cdot \left(\frac{C_f}{2}\right)^{\frac{1}{2}} \cdot \left(Pr_f^{\frac{2}{3}} - 1\right)} \cdot \left(\frac{Pr_f}{Pr_{ri}}\right)^{0,11} \quad (3.16)$$

- U_{rec} , coeficiente de transmisión de calor al interior. Viene dado por la ec.(2.4) comentada previamente.

La Clase HCE también nos proporciona algunos métodos necesarios para el trabajo de procesamiento de la información, asignación y recuperación de valores de los atributos. Otro aspecto importante es que cada instancia de la clase HCE tiene un atributo de tipo *diccionario* denominado **parameters** en el que, a modo de lista de pares clave-valor, va a recibir aquellos parámetros que posteriormente serán empleados por el Modelo. Los diferentes autores que han elaborado modelos para los HCE no siempre utilizan los mismos parámetros ni idénticos identificadores. Al emplear un diccionario se facilita la tarea de implementación de nuevos Modelos, sin que sea necesario cambiar los atributos de la clase en cada ocasión. Entre los parámetros que se pasan al HCE durante la creación de su instancia están su absorbividad solar α_{solar} , la transmisividad del vidrio τ y su reflectividad ρ .

En el caso de la planta simulada se ha utilizado un HCE fabricado por Solel cuyos parámetros se guardan en una librería en formato JSON. Parte de los datos de cada HCE de la librería se han extraído de los archivos de configuración de SAM (System Advisor Model), el software de referencia para la simulación de plantas de energías renovables. No obstante, el modelo no hace uso de todos ellos y, en cambio, se precisa de algún dato más para realizar la simulación. Estos parámetros son almacenados en el diccionario *parameters*. El programa desarrollado permitiría, en principio, modelar cada HCE con unos parámetros diferentes, es decir, que cada HCE se comportase de forma diferente al resto. Esta funcionalidad puede ser interesante para el estudio de comportamiento del campo solar cuando se dispone de estadísticas adecuadas sobre cómo evoluciona en el tiempo y se distribuye en el campo solar cada parámetro, lo cual hace que no todos los lazos se comporten de igual manera. El inconveniente es que el tiempo de cálculo aumenta notablemente al tener que simularse cada lazo independientemente.

3.2.3. Clase SCA, para el modelado del Solar Collector Assembly

Un SCA es una estructura compuesta por una serie de reflectores que concentran la radiación solar sobre los HCE. Desde el punto de vista operativo, un SCA cuenta con capacidad de movimiento independiente respecto al resto de SCAs de la planta, por lo que es la unidad mínima de control de enfoque o desenfoque de la radiación solar en el campo solar. La clase SCA nos permite modelar cada SCA teniendo en cuenta las propiedades de los reflectores (reflectividad, suciedad de los espejos, precisión del movimiento de seguimiento solar, etc.)

En plantas CCP dedicadas a maximizar la generación anual de energía eléctrica, el sistema de seguimiento tiene su eje de rotación alineado en la dirección Norte-Sur con el fin de hacer un seguimiento Este-Oeste de la trayectoria solar a lo largo del día. No obstante, una configuración con eje Este-Oeste también puede ser interesante en algunos casos y el modelo también permite esta configuración.

Dentro del campo solar, cada HCE debe pertenecer a un SCA. El primer HCE del SCA recibe el fluido caloportador procedente de otro SCA o de las tuberías colectoras de HTF frío. El último HCE del SCA entrega el HTF más caliente al siguiente SCA o a las tuberías colectoras de HTF caliente a la salida del lazo.

El SCA cuenta con un método para el cálculo del modificador por ángulo de incidencia. Hemos considerado que el SCA mantendrá en todo momento un ángulo de seguimiento β óptimo con el fin de minimizar el ángulo de incidencia. En este caso, el *IAM* se calcula según la expresión dada por la ec.(3.17)

$$IAM = F_0 + F_1 \cdot \frac{\theta}{\cos(\theta)} + F_2 \cdot \frac{\theta^2}{\cos(\theta)}, \quad \forall \theta \in (0^\circ, 80^\circ) \quad (3.17)$$

Algunos fabricantes incluyen un factor $\cos(\theta)$ en la expresión del *IAM*, por lo que no deberá incluirse entonces en la ecuación del rendimiento total del HCE. En nuestro caso, para el UVAC 3 de Solel empleado en la simulación, no es así.

Otro valor que debe ofrecernos la clase que modela al SCA es la *fracción solar* o *factor de interceptación*, que permite estimar la tasa entre la radiación solar que alcanza al reflector y la que posteriormente incide realmente sobre el tubo absorbedor. Su valor se

obtiene según la ec.(3.18) como producto de una serie de factores:

$$\gamma = \eta_{geomtrico} \cdot \eta_{seguidor} \cdot \eta_{suciedad} \cdot \eta_{disponibilidad} \quad (3.18)$$

El factor geométrico $\eta_{geomtrico}$ depende de las imperfecciones geométricas de conjunto reflector-absorbedor como pequeñas desviaciones en la curvatura de los espejos o la deformación de la estructura. El factor de precisión del seguidor $\eta_{seguidor}$ permite considerar los errores de seguimiento del mecanismo de movimiento del reflector. El factor de suciedad $\eta_{suciedad}$ se refiere a la pérdida de reflectividad debida a acumulación de polvo en los espejos. Se ha considerado que esta acumulación de polvo afecta también a la transmisividad de la cubierta de vidrio del HCE, por lo que este factor se computa dos veces. Finalmente, el factor de disponibilidad $\eta_{disponibilidad}$ considera las pérdidas que ocasionalmente se puedan producir por averías del sistema de concentración.

El SCA, como sistema responsable del seguimiento solar, también cuenta con un método para ofrecernos información sobre el ángulo de incidencia β en el plano de apertura del reflector. Las expresiones generales pueden encontrarse en [18] pero, en nuestro caso, hemos recurrido a la librería *pvlib-python* [6], desarrollada en *Sandia National Laboratories* con el objetivo de impulsar el desarrollo de herramientas de código abierto para la simulación de sistemas fotovoltaicos. Concretamente, nos valemos de este código para obtener los valores del ángulo de incidencia y la posición solar para cada fecha del año según las coordenadas geográficas del lugar donde se realiza la simulación.

3.2.4. Clase Loop

Un Loop o lazo es un conjunto de SCAs conectados en serie de tal forma que el HTF que entra frío al lazo experimenta un salto térmico cuando transita por él. El sistema de control ajusta el estado de enfoque o desenfoque de cada SCA en el lazo con el fin de conseguir que la temperatura de salida sea la de consigna. Por motivos de económicos, el caudal de HTF no suele ser regulable a nivel de lazo, pues obligaría a instalar una válvula de control en cada uno de ellos, por lo que todos los lazos de un mismo subcampo suelen tener un caudal muy parecido. Se ha desarrollado también una Clase Subfield para dar cuenta del conjunto de lazos que pertenecen a un mismo subcampo y, por tanto, pueden variar su caudal de forma independiente de los lazos de otro subcampo. En un campo solar

suele haber haber varios subcampos que pueden regular su caudal independientemente. Los casos más extremos serían el de un campo solar con un único subcampo, en el que todos los lazos pertenecen al mismo subcampo y en el otro extremo, un campo solar con tantos subcampos como lazos tiene, de tal forma que cada lazo es el único en su subcampo y por tanto cada lazo puede regular su caudal independientemente.

Cada lazo del campo solar es modelado mediante una instancia de la clase Loop. Cada instancia mantiene referencias al subcampo al que pertenece y a los SCA que contiene. Atributos importantes de estas instancias serán el caudal de HTF en el lazo, las temperaturas de entrada y salida del HTF y el rendimiento completo del lazo.

El código permite trabajar de dos formas:

- Método `calc_loop_pr_for_tout`, que calcula el caudal requerido con el que conseguir una temperatura de HTF determinada.
- Método `calc_loop_pr_for_massflow`, que mantiene fijo el caudal de HTF que se le indica y calcula qué temperatura de salida alcanzará el HTF.

En caso de que esta temperatura de salida supere el máximo permitido (un valor configurable) se producirá un desenfoque en el SCA en que se alcance esta temperatura y el HTF dejará de calentarse. En este caso se suele decir que se produce un vertido de energía o desaprovechamiento de la radiación existente. El código permite contabilizar esta energía desaprovechada por cada lazo durante estas situaciones.

La Clase BaseLoop hereda de la Clase Loop sus principales métodos y atributos pero supone una pequeña variación de una instancia o lazo definido por Loop ya que se trata de un lazo "típico" o "promedio", que presenta una configuración constructiva idéntica a la del resto de lazos de la planta pero no pertenece a ningún subcampo solar. Este lazo especial o prototipo se empleará cuando queramos realizar un estudio paramétrico del comportamiento del lazo, no de la planta, y también para realizar una simulación mucho más rápida cuando asumamos la hipótesis de que todos los lazos de la planta se comportan de igual manera. Esta aproximación es la que se hace en aplicaciones como SAM, donde no se simulan todos los lazos para modelar el campo solar, sino que se simula solo un lazo y el resultado se obtiene multiplicando el caudal de salida de este lazo por el número de lazos que forman la planta.

3.2.5. Clase Subfield, modelado del subcampo

Un subcampo es un conjunto de lazos en los que se considera que el HTF se repartirá equitativamente. Cada subcampo dispone de una válvula de control de caudal a su entrada y representa el mayor grado de control de caudal de HTF que se puede alcanzar en el campo solar. Un campo solar suele contar con varios subcampos y cada uno de ellos, a su vez, cuenta con varios lazos.

La Clase Subfield mantiene referencias a lo lazos que lo constituyen, es decir, a cada una de las instancias que representan un lazo. También mantiene referencia al campo solar al que pertenece y cuenta con métodos para calcular cual será la temperatura de salida del HTF del subcampo solar una vez que el caudal de salida de cada lazo se haya mezclado con el del resto de lazos. Nótese que aunque se supone que todos los lazos del subcampo tienen el mismo caudal másico la temperatura de salida de cada uno de ellos puede ser diferente y, por tanto, la energía aportada por cada lazo también lo será.

3.2.6. Clase Solarfield, modelado del campo solar

El campo solar alberga el conjunto de subcampos y, por tanto, todos los lazos de la instalación. Se trata del objeto que en última instancia queremos modelar con el fin de conocer cómo será el comportamiento de la planta solar y su rendimiento anual. A la hora de definir el campo solar para el modelo es necesario conocer cuántos subcampos contiene, cuántos lazos hay en cada subcampo, qué configuración tiene cada lazo (número de SCAs en cada lazo y número de HCEs en cada SCA). También es importante conocer la distancia entre lazos con el fin de estimar el sombreado que se produce a primera y última hora del día.

El Clase Solarfield también recibe una serie de valores nominales para los que se ha realizado un diseño óptimo del sistema, como las temperaturas y presiones de entrada y salida del HTF, las temperaturas máximas y mínimas tolerables por razones de seguridad, el caudal nominal (normalmente se suele dar este caudal por lazo y el caudal del campo solar será la suma de todos ellos), el caudal mínimo (existe cierta limitación tanto por la velocidad mínima de las bombas como por otras cuestiones operativas que desaconsejan que el HTF circule por debajo de este límite).

Cuando se crea una instancia de la Clase **Solarfield**, ésta recibe los datos de configuración que se han pasado para la simulación y lanza el proceso de construcción que genera, en base a esa configuración, los diferentes subcampos, lazos, SCAs y HCEs que forman el campo solar.

3.2.7. Clase **Fluid y sus clases hijas, **FluidCoolProp** y **FluidTabular****

Para modelar el HTF (Heat Transfer Fluid) se ha creado la Clase **Fluid**. Las propiedades del HTF pueden obtenerse mediante funciones polinómicas con coeficientes constantes calculados experimentalmente o desde librerías preexistentes como *CoolProp*. Por ese motivo, según se opte por un método u otro, se han creado las subclases **FluidCoolProp** y **FluidTabular**. No obstante, se ha comprobado que el número de fluidos existentes en la librería **FluidCoolProp** que suelen emplearse como HTF no es muy grande, limitándose a *Therminol VP-1* y *Syltherm 800*. No se encuentra en esta librería el aceite *Dowtherm-A*, que es el que se emplea en la planta cuyos datos se han empleado para el desarrollo de esta herramienta. Pero el mayor inconveniente reside en que *CoolProp* devuelve valores solo dentro del rango de temperaturas de uso válidas según el fabricante.

Este rango es demasiado estricto y se producen problemas debido a la devolución de valores no numéricos, especialmente cuando se está calculando la temperatura teórica de salida del HTF en condiciones de sobrecalentamiento. Por este motivo, se ha hecho uso mayoritariamente de la clase **FluidTabular**, con funciones polinómicas con coeficientes calculados a partir de los datos ofrecidos por los fabricantes. La clase **Fluid** y sus clases hijas ofrecen métodos para calcular la densidad, viscosidad cinemática, número de Reynolds, calor específico, conductividad térmica y entalpía en función de la temperatura y la presión. También ofrecen un método para calcular la temperatura del fluido en función de la entalpía y la presión, considerando entalpía cero para líquido saturado según ASHRAE a una temperatura de 285,856 K.

Para el modelado sería necesario conocer la dependencia de los parámetros citados en función de la temperatura y la presión, pero se ha comprobado que la variación de estos parámetros con la presión es despreciable cuando se trabaja en condiciones de líquido saturado. Ya que en una planta termosolar se debe trabajar siempre con esta premisa, estando los circuitos de HTF siempre bien presurizados, expresaremos dichos parámetros a

partir de un polinomio de mayor o menor grado, en función exclusivamente de la temperatura. También se ha obtenido una curva para calcular la temperatura del líquido saturado en función de su entalpía.

- $\rho(T)$: Densidad, (kg/m^3)
- $\mu(T)$: Viscosidad dinámica, ($Pa \cdot m$)
- $k_T(T)$: Conductividad térmica, ($W/m \cdot K$)
- $c_p(T)$: Calor específico a presión constante, ($J/kg \cdot K$)
- $h(T)$: Entalpía específica, (J/kg), considerando $h=0$ a $T_{ref} = 285,86K$
- $T(H)$: Temperatura en función de la entalpía, (K)

La fórmula general para cada uno de estos parámetros es del tipo de la ec.(3.19):

$$parametro(T) = a_0 + a_1 \cdot T + a_2 \cdot T^2 + a_3 \cdot T^3 + a_4 \cdot T^4 + a_5 \cdot T^5 + a_6 \cdot T^6 + a_7 \cdot T^7 + a_8 \cdot T^8 \quad (3.19)$$

No todos los parámetros requieren un polinomio de grado tan alto y los coeficientes que acompañan a los términos de mayor grado son nulos, pero se ha empleado esta estructura por homogeneizar el proceso de carga de datos.

Para el caso del *Dowtherm-A* se han obtenido los coeficientes de los polinomios que caracterizan cada parámetro a partir de [24] y se han contrastado las curvas con los datos ofrecidos en [25]. En el caso de la viscosidad cinemática se ha detectado que el polinomio de 8º grado que se obtiene con los coeficientes de la primera referencia presenta una gran desviación y crecimiento asintótico para temperaturas ligeramente superiores a la máxima de operación del fluido. Con el fin de poder flexibilizar el proceso de cálculo y que no se produzcan desbordamientos se ha ajustado un nuevo polinomio tras extender los datos de la viscosidad dinámica hasta unos 450 °C aproximadamente según la tendencia observada en el último tramo de la curva $\mu(T)$. De esta forma se obtiene un nuevo polinomio, con mejor comportamiento en este rango extendido.

En las figuras 3.3 puede verse el comportamiento estos parámetros en función de la temperatura para el caso del *Dowtherm-A*.

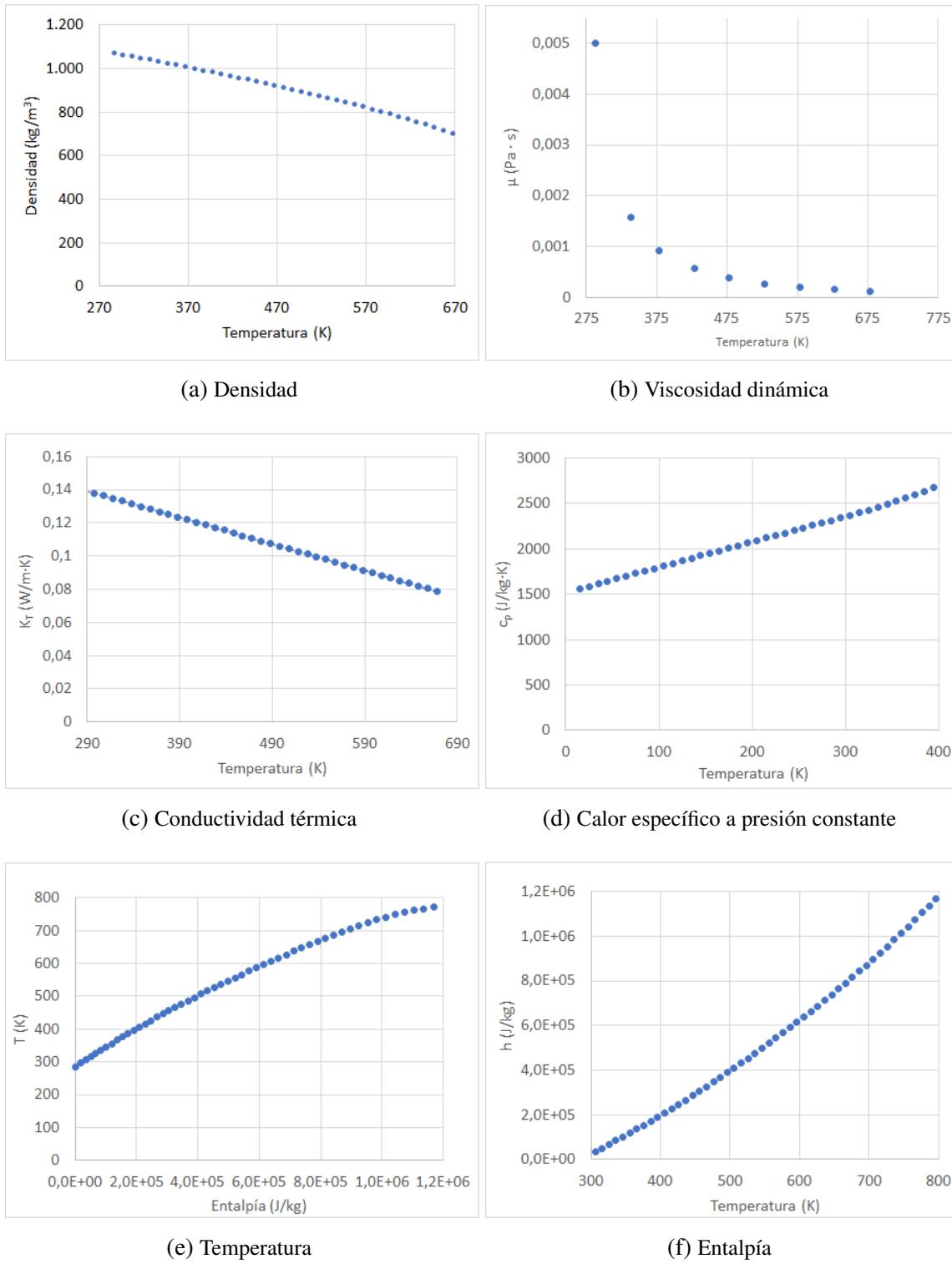


Fig. 3.3. Parámetros físicos del *Dowtherm-A* en estado de líquido saturado. Algunos puntos se han extrapolado más allá de la temperatura máxima de trabajo con el fin de evitar errores durante los procesos de convergencia.

Los coeficientes de los polinomios de ajuste para *Dowtherm-A* se muestran en la tabla 3.1:

TABLA 3.1. Coeficientes polinómicos para el fluido caloportador
Dowtherm-A

Parámetro	$C_p(T)$	$\mu(T)$	$\rho(T)$	K_T	h(T)	T(h)
a_0	-2,36E+03	1,58E+00	1,49E+03	1,86E-01	-6,51E+05	2,85E+02
a_1	3,95E+01	-2,34E-02	-3,33E+00	-1,60E-04	4,12E+03	6,21E-04
a_2	-1,70E-01	1,50E-04	1,25E-02	5,91E-12	-1,24E+01	-1,82E-10
a_3	3,90E-04	-5,49E-07	-2,97E-05	0,00E+00	2,77E-02	-1,42E-16
a_4	-4,42E-07	1,24E-09	3,44E-08	0,00E+00	-2,78E-05	3,32E-22
a_5	1,98E-10	-1,78E-12	-1,62E-11	0,00E+00	1,11E-08	-1,75E-28
a_6	0,00E+00	1,58E-15	0,00E+00	0,00E+00	0,00E+00	0,00E+00
a_7	0,00E+00	-7,94E-19	0,00E+00	0,00E+00	0,00E+00	0,00E+00
a_8	0,00E+00	1,73E-22	0,00E+00	0,00E+00	0,00E+00	0,00E+00

El mismo procedimiento se ha empleado para obtener los polinomios característicos del fluido *Therminol VP-1*, aunque en este caso los polinomios se han ajustado a partir de una lista de valores sacada de la librería *CoolProp*. Pese a que el programa de simulación permite que durante el tiempo de ejecución se obtengan los parámetros citados a través de esta librería, existe una limitación debido al rígido margen de temperaturas con el que esta librería trabaja para cada fluido y esto provoca que, para valores de temperatura ligeramente superiores al rango de operación ofrecido por fabricante, no se devuelva ningún valor. Esto resulta en problemas en tiempo de ejecución si algún lazo alcanza una temperatura superior a los 397°C (398°C en el caso de Syltherm 800). Aunque superar esta temperatura no es recomendable, es algo que puntualmente ocurre durante la operación de la planta. Además, una forma calcular la energía desaprovechada por desenfoque sería a partir de la temperatura que hubiera alcanzado el lazo de no haberse producido el desenfoque y calculando posteriormente su entalpía. Esta aproximación, que sería imposible en la vida real debido a la degradación del HTF e incluso al daño del propio sistema por las sobrepresiones que se producirían, facilita el cálculo de la energía desaprovechada en cada momento. Se ha supuesto que las curvas de los parámetros se mantienen bien ajustadas siempre y cuando la sobretemperatura alcanzada no sea excesiva (en las simulaciones

realizadas no se ha superado más del 10 % de la temperatura máxima de operación recomendada por el fabricante). Por estos motivos, para este trabajo se han empleado siempre los valores de los parámetros obtenidos a partir de los polinomios y no de *CoolProp*.

Para el caso del *Therminol VP-1* los coeficientes calculados son los que se muestran en la tabla 3.2:

TABLA 3.2. Coeficientes polinómicos para el fluido caloportador *Therminol VP-1*

Parámetro	$C_p(T)$	$\mu(T)$	$\rho(T)$	K_T	$h(T)$	$T(h)$
a_0	2,881E+02	1,487E+00	1,403E+03	1,486E-01	-2,923E+05	2,924E+02
a_1	5,875E+00	-2,186E-02	-1,613E+00	9,755E-06	3,910E+02	6,424E-04
a_2	-6,857E-03	1,400E-04	2,138E-03	-1,780E-07	2,076E+00	-3,396E-10
a_3	4,844E-06	-5,092E-07	-1,931E-06	3,524E-12	1,811E-03	2,587E-16
a_4	6,960E-20	1,148E-09	-9,610E-21	-7,572E-25	-1,089E-05	-1,066E-22
a_5	-2,780E-23	-1,642E-12	3,864E-24	2,948E-28	2,274E-08	0,000E+00
a_6	0,000E+00	1,454E-15	0,000E+00	0,000E+00	-2,667E-11	0,000E+00
a_7	0,000E+00	-7,286E-19	0,000E+00	0,000E+00	1,788E-14	0,000E+00
a_8	0,000E+00	1,583E-22	0,000E+00	0,000E+00	-5,284E-18	0,000E+00

3.2.8. Clases **Weather**, **FieldData** y **TableData**

Estas clases implementan métodos adecuados para la adquisición de datos desde diferentes tipos de ficheros, en concreto:

- Clase **Weather** para ficheros .tmy con datos meteorológicos (*Weather Files*). En estos ficheros solo hay datos meteorológicos como radiación normal incidente (DNI), temperatura de bulbo seco, y datos geográficos del emplazamiento (Site), como latitud, longitud y altitud. A partir de estos datos se pueden realizar simulaciones para ver cuál sería el comportamiento de la planta con estas condiciones.
- Clase **FieldData** para ficheros .csv con datos recogidos de alguna planta (*Field Data Files*). Estos ficheros contienen datos meteorológicos recogidos por las estaciones de planta y también datos de instrumentación de planta, en concreto, caudal-

les, temperaturas y presiones de entrada y salida del campo solar y de los diferentes subcampos que puedan existir. Los encabezados de cada columna probablemente serán identificadores o *tags* propios de cada planta, por lo que es necesario indicar al programa a qué dato corresponde cada *tag*. Esto se puede hacer en el fichero de configuración de la simulación. Con estos datos se puede simular el comportamiento del campo solar para el caudal teórico requerido pero también comprobar cuál sería el rendimiento esperado del campo solar operando con el caudal real de planta. Los datos obtenidos podrán después compararse con los reales de funcionamiento de planta. A este tipo de simulaciones las denominaremos *benchmark*.

- Clase `TableData` para ficheros .csv empleados en otro tipo de simulaciones, por ejemplo para el análisis paramétrico o para el estudio del rendimiento de un lazo en función de diferentes valores de \dot{q}_{abs}'' .

3.2.9. Clase Site

La Clase `Site` (emplazamiento), contiene la información relativa al lugar donde está ubicada la planta. Los datos de latitud, longitud y altitud son importantes a la hora de calcular la trayectoria solar para cada fecha. Nos ofrece un método para calcular la posición del sol en cada fecha del año en base a los parámetros que almacenan las coordenadas geográficas.

Esta clase cuenta también con el método `get_solarposition`, que mediante una llamada a la función `pvlib.solarposition.get_solarposition` de la librería `pvlib-python` nos permite obtener información relativa a la posición solar para cada fecha del año. Esta librería es una implementación en *Python 3* del algoritmo *Solar Position Algorithm*, SPA, desarrollado por el *National Renewable Energy Laboratory* (NREL) para el cálculo de la posición solar para cualquier fecha y coordenadas geográficas entre los años 2000 a. C. y 6000 d. C., con una incertidumbre de $\pm 0,0003^\circ$ [26]. Este algoritmo ha sido testado en innumerables aplicaciones en todo el mundo. Los datos de la implementación en *Python* que nosotros hemos obtenido son compatibles con los que se obtienen en SAM, donde la implementación es en *C++*.

3.3. Procedimiento para realizar una simulación

En este apartado se describe cómo puede desarrollarse un *script* o programa, a partir de las Clases ya implementadas y comentadas anteriormente, con el fin de realizar diferentes tipos de simulaciones. El procedimiento es flexible y aquí tan solo se dan algunos apuntes sobre cómo se aprovechan las estructuras ya creadas. Se continua con la filosofía de POO y se hace uso de una clase denominada `SolarFieldSimulation`. Esta clase tiene una función meramente práctica, pues implementa una serie de métodos que permiten crear la estructura del campo solar y las instancias de las clases necesaria para poder lanzar la simulación. De esta forma, evitamos tener que escribir todo ese código en un programa cada vez que queramos ejecutar una simulación de un campo solar. La clase cuenta con métodos para leer secuencialmente el archivo con los datos, ejecutar cada tipo de simulación, recopilar los datos agregados y guardar y mostrar los resultados.

El objetivo que nos proponemos es simular el comportamiento de un campo solar bajo unas determinadas condiciones. Puesto que estas condiciones varían a lo largo del día, se emplearán ficheros de datos en formato tabular que cuentan con una columna índice para la fecha y hora indicadas. Con el fin de poder reutilizar el trabajo realizado durante el trabajo de configuración de la simulación, se emplea un archivo en formato JSON que recoge todos los parámetros necesarios. En resumen, la instancia de `SolarFieldSimulation` realiza los siguientes pasos:

- Lee el archivo de configuración de la simulación y almacena los parámetros necesarios.
- Se crea una instancia de la clase `Site` con información sobre la ubicación de la planta.
- Se crea una instancia para el almacenamiento de los datos del fichero en formato tipo tabla. En función de si el fichero es de tipo meteorológico (TMY2 o TMY3) o es un fichero en formato CSV creará una instancia de la clase `Weather` o `FieldData` respectivamente. Los datos cargados se almacenan en un `DataFrame` de la librería `Pandas` denominado *datasource*.
- Para el modelado del HTF, si las propiedades del fluido se van a tomar desde la

librería externa *CoolProp*, se crea una instancia de la Clase `FluidCoolProp`. Si las propiedades se van a obtener mediante polinomios característicos del fluido, se crea una instancia de la Clase `FluidTabular`.

- Se crea una instancia `SolarField` a partir de los parámetros de configuración de campo solar. Al crearse esta instancia, se genera, en base a los parámetros que se le pasan, todo la estructura de subcampos, lazos, SCAs y HCEs del campo.
- Se crea una instancia de la clase `BaseLoop` a partir de los parámetros de configuración de lazo.

A partir de aquí, la instancia de `SolarFieldSimulation` ya dispone de lo necesario para realizar la simulación del campo mediante su método `runSimulation`.

El tipo de simulación que se realiza depende del tipo de datos de que se disponga y de lo que se seleccione en el archivo de configuración. Existen dos tipos de simulación posibles:

- Simulación tipo *simulation*: En este caso, el caudal del lazo se recalculará en un proceso de convergencia hasta conseguir que la temperatura de salida del lazo sea la temperatura consignada.
 - Si el tipo de datos del que se dispone no tiene datos reales de planta, la instancia `datasource` será de tipo `Weather` y solo se cuenta con datos meteorológicos de (DNI), temperatura ambiente, (T_{ext}), velocidad del viento (W_{spd}) y presión atmosférica (pressure), por lo que la temperatura de entrada a los lazos será la nominal.
 - Si el tipo de datos del que se dispone sí cuenta con datos reales de planta que permitan conocer las temperaturas de entrada a los lazos (como es nuestro caso), la simulación puede usar estas temperaturas a la hora de ajustar los caudales al salto térmico necesario.
- Simulación tipo *benchmark*: En este caso se debe disponer obligatoriamente de datos reales de planta, pues la simulación utiliza las temperaturas de entrada a los lazos y los caudales reales para calcular cuál será la temperatura de salida. Posteriormente, en los archivos de salida de datos, se puede comparar la temperatura real

de salida del lazo con la calculada y de esta forma estimar si ha habido desenfoque y por tanto, desaprovechamiento de la energía solar. Hay que tener en cuenta en los datos que disponemos podemos encontrar situaciones en las que el lazo alcanza su temperatura de consigna pero es posible que se estuviera realizando un ajuste de enfoque-desenfoque (ya que el caudal no es regulable a nivel de lazo). En ese caso, es interesante saber qué temperatura hubiera alcanzado el HTF de no haberse producido el desenfoque y, por tanto, poder calcular la energía solar que no se ha aprovechado por no poder introducir mayor caudal en el lazo.

A la hora de realizarse cada una de estas simulaciones, puede darse el caso de que se haya configurado la opción *fastmode=True*. En este caso se considera que todos los lazos de la planta se comportan como el lazo típico, modelado mediante la Clase BaseLoop. En caso contrario, la simulación se realizará para cada lazo del campo, lo que puede ser interesante si los lazos o sus componentes cuentan con diferentes valores en sus parámetros. Se ha implementado esta posibilidad de cara a desarrollos futuros.

Una vez que el método `runSimulation` ha procesado todas las filas seleccionadas del DataFrame *datasource*, los datos calculados que se han ido añadiendo al DataFrame son volcados a un archivo CSV para su posterior análisis.

En la Fig.3.4 se muestra, de manera esquemática, cómo una instancia de la Clase `SolarFieldSimulation` recibe el archivo de configuración creado mediante el programa auxiliar *interface.py* y, a partir de esta configuración, lanza la simulación del campo solar leyendo los datos del fichero que se haya especificado.

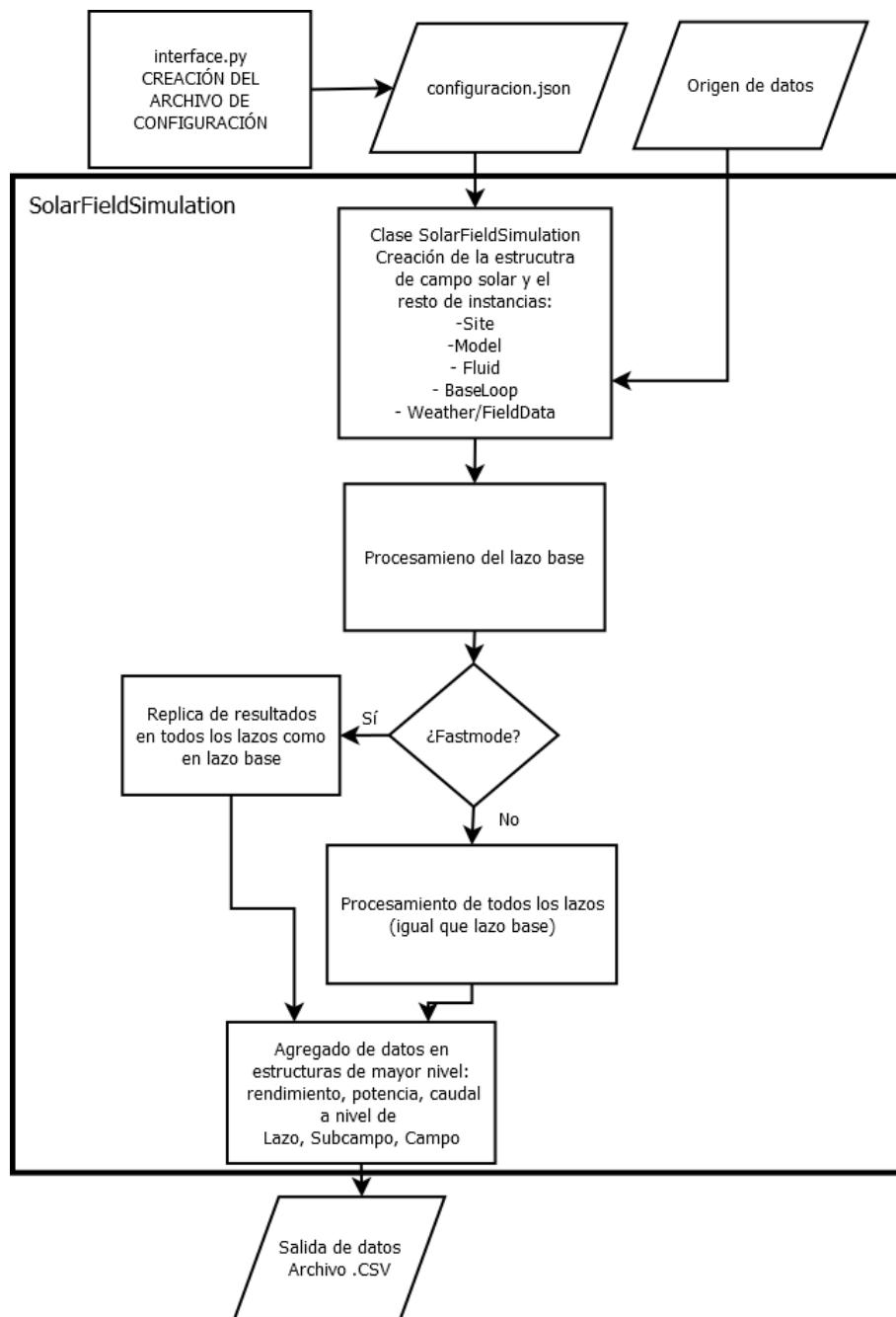


Fig. 3.4. Esquema de operación global de una simulación de un campo solar mediante la clase **SolarFieldSimulation**

Según el tipo de simulación, durante el procesamiento del lazo se sigue o no un proceso de ajuste de caudal para conseguir obtener a la salida del lazo una temperatura determinada. El diagrama de flujo desarrollado para este bloque puede verse en la Fig.3.5. En una misma ejecución del programa pueden ejecutarse ambas o solo una de las simulaciones.

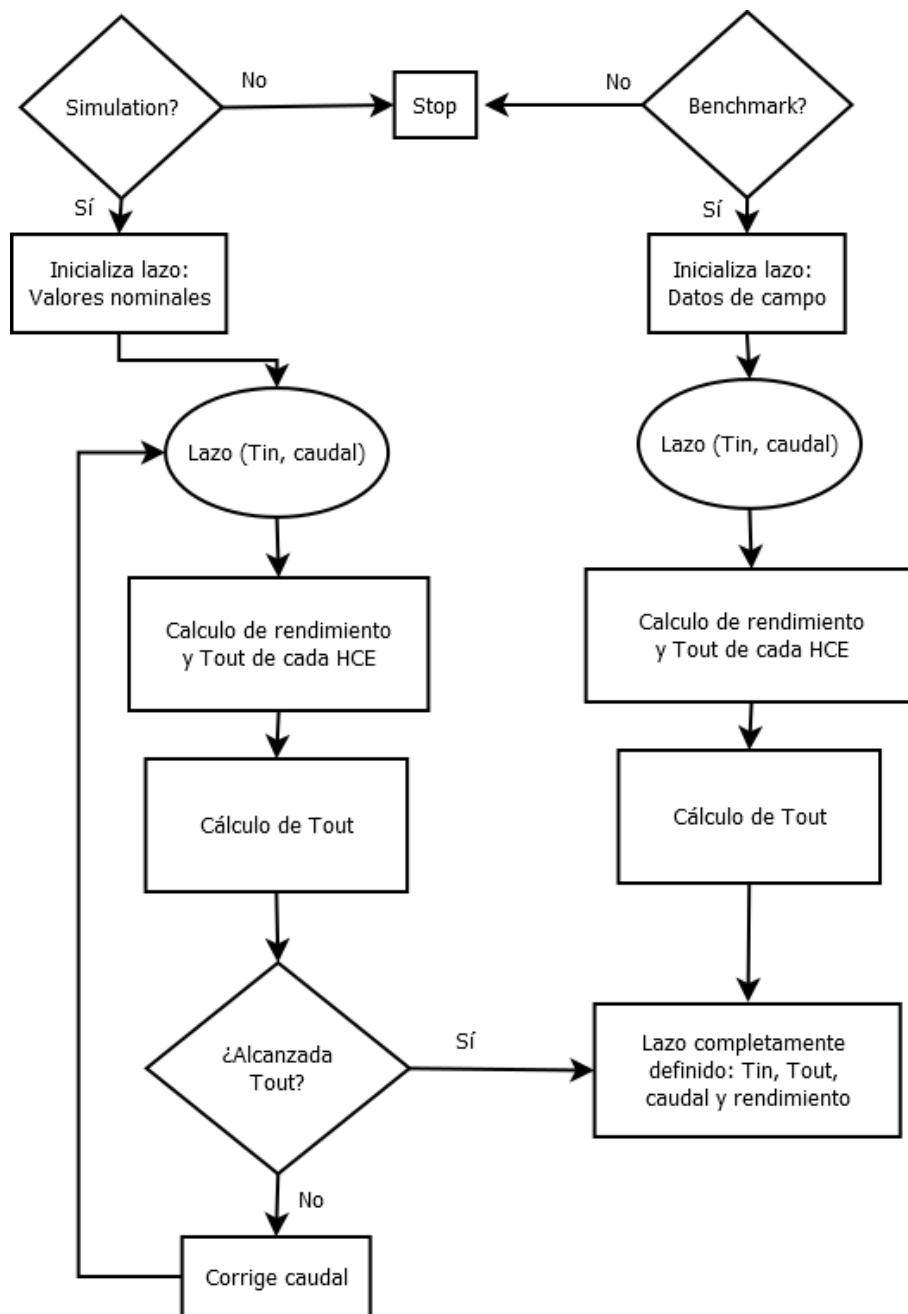


Fig. 3.5. Esquema desarrollado del bloque de *procesamiento del lazo* para las simulaciones tipo *simulation* y *benchmark*

3.4. Verificación por comparación con otra herramienta de simulación

Con el fin realizar una primera verificación de nuestro código, compararemos en primer lugar los valores obtenidos para una determinada configuración de campo con los que se obtienen mediante System Advisor Model (SAM, [8]), un software de reconocido prestigio muy empleado en el sector de las energías renovables. Las posibilidades de SAM

van mucho más allá del alcance de nuestro código, pudiendo realizar el modelado no solo de sistemas de energía solar de concentración sino también de sistemas fotovoltaicos, geotérmicos, mareomotrices, eólicos, de biomasa por ejemplo. Dentro de los sistemas de energía solar térmica de concentración, ofrece la posibilidad de modelar centrales de las cuatro principales tecnologías ya comentadas y, a su vez, de diversas variedades dentro de cada una de éstas. SAM realiza también el análisis económico y financiero del proyecto basándose principalmente en la generación de electricidad que se espera producir. Por este motivo, SAM tiene en cuenta el acoplamiento del campo solar con el bloque de potencia a la hora de realizar la simulación. Esto hace que no podamos comparar directamente nuestro programa con los modelos de SAM para generación eléctrica, pues los ajustes que hace SAM afectan al comportamiento del campo, introduciendo limitaciones de caudal de HTF y de potencia térmica transferida al bloque de potencia, desenfoques en el campo, rampas de arranque e inercias del sistema que quedan fuera de nuestro alcance. En cambio, SAM también cuenta con un módulo para la simulación de un sistema de generación de calor para proceso industrial (IPH, Industrial Process Heat) basado en el modelo Físico que emplea para la simulación de plantas CCP. Emplearemos este módulo para comparar los resultados de SAM con nuestro simulador tal y como ese explica a continuación.

3.4.1. Configuración de la simulación para comparación con SAM

Para que una simulación realizada con el módulo IPH de SAM sea comparable a la nuestra, indicaremos a SAM que no existe almacenamiento térmico para evitar los procesos de mezcla de HTF a diferentes temperaturas que se producirían en ese caso. También indicaremos que el sumidero térmico tiene capacidad elevada, lo que, en principio, nos obligaría a seleccionar un tamaño de campo solar muy grande. Para que el tamaño de campo solar sea igual al del campo que queremos simular indicaremos a SAM que las condiciones de radiación solar nominales son mucho más elevadas que las que se alcanzarán en cualquier momento el año (hemos puesto un valor de $(2000W/m^2)$). De este modo, conseguimos que el dimensionado que hace SAM del campo solar sea igual que el nuestro (120 lazos), pero que a lo largo de la simulación con datos de radiación reales no se produzca desenfoque en ningún momento. Es decir, con esta configuración de SAM extraemos del campo solar toda energía posible, al igual que hace nuestro simulador.

Puesto que solo modelamos el campo, no tenemos a priori ninguna información sobre la temperatura de retorno del HTF frío desde el intercambiador donde cede su energía al sumidero térmico. Con el fin de que la comparación sea lo más ajustada posible, aprovecharemos los datos de temperatura de HTF frío que se obtienen de la simulación de SAM como datos de entrada para nuestra simulación. De esta forma, en ambas simulaciones el salto térmico se realiza desde el mismo punto de partida. Además, indicaremos a SAM que no existe inercia térmica en tuberías y concentradores y prescindiremos del cálculo de pérdidas térmicas en las primeras.

Respecto al fluido caloportador, SAM nos permite seleccionar entre una serie de fluidos preconfigurados. Seleccionaremos *Therminol VP-1* pues también podemos simular este fluido con nuestro programa.

Realizaremos la simulación para un año completo. Contamos con los datos meteorológicos del año 2007 en formato TMY3. En la Fig.3.6 se puede ver la pestaña de configuración de SAM para seleccionar el origen de datos meteorológicos.

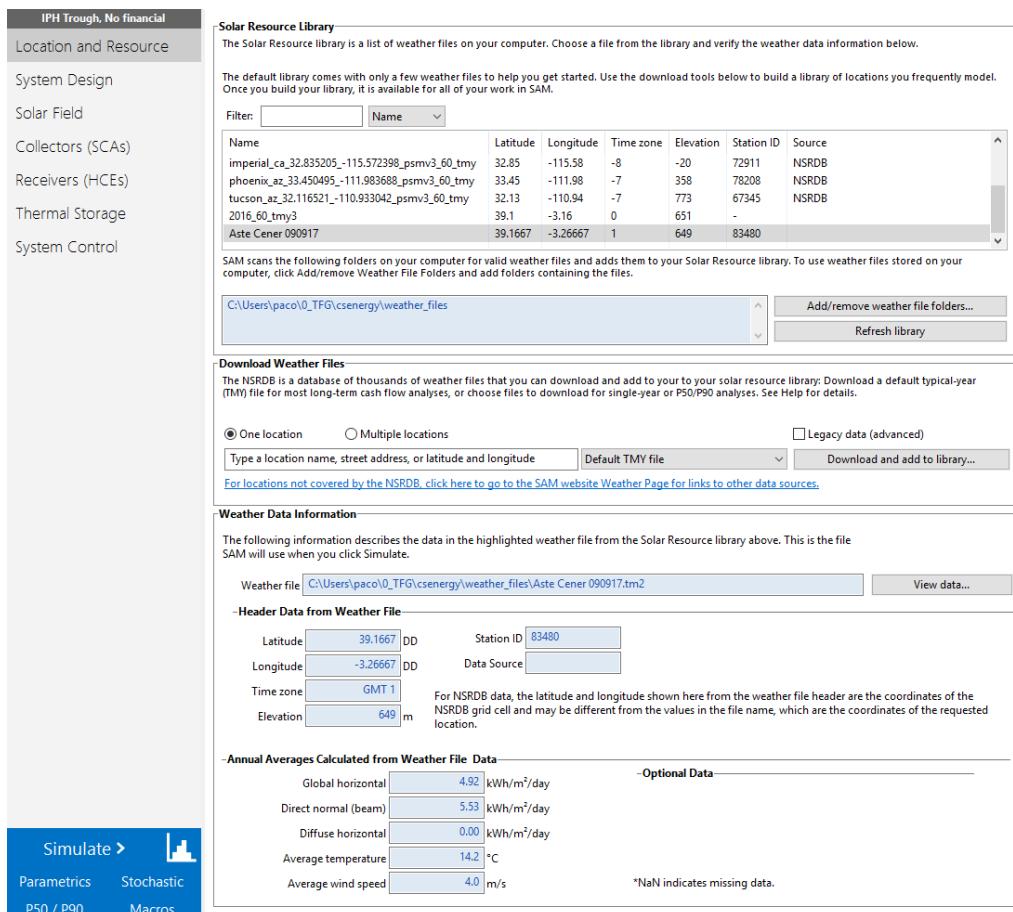


Fig. 3.6. Configuración SAM. Selección del archivo de datos meteorológicos

La versión de SAM utilizada ha sido la "2020.2.29, 64 bit, updated to revision 1". En la Fig.3.7 podemos ver la pestaña de configuración del campo solar.

The screenshot shows the SAM software's configuration interface for a solar field. The left sidebar lists categories: IPH Trough, No financial, Location and Resource, System Design, Solar Field (selected), Collectors (SCAs), Receivers (HCEs), Thermal Storage, and System Control. The main area has several sections:

- Design Point Parameters**: Includes fields for Design point DNI (2520 W/m²), Target solar multiple (2.5), Target receiver thermal power (750.00 MWt), Loop inlet HTF temperature (293 °C), Loop outlet HTF temperature (393 °C).
- Heat Sink**: Includes fields for Heat sink power (300.00 MWt), Pumping power for HTF through heat sink (0.55 kW/kg/s), Model piping through heat sink? (unchecked), Length of piping through heat sink (0.0 m), and a "Choose Number of Loops" button.
- Thermal Energy Storage**: Includes a field for Hours of storage at design point (0 hours).
- System Summary**: Includes fields for Actual number of loops (120), Actual solar multiple (2.51), Total aperture reflective area (392,400.0 m²), Actual field thermal output (752.41 MWt).

Fig. 3.7. Configuración SAM. Campo solar

Los SCA, modelo SenerTrough I diseñado por Sener, se han parametrizado de la siguiente manera a partir del modelo EuroTrough ET150:

TABLA 3.3. Configuración del SCA modelo SenerTrough I de Sener

Parámetro	Valor
Longitud	148,5 (m)
Apertura	5,77 (m)
Distancia Focal	2,1 (m)
Coeficiente F_0 del IAM	1
Coeficiente F_1 del IAM	0,0506
Coeficiente F_2 del IAM	-0,1763
Precisión del seguidor	0,99
Precisión geométrica	0,98
Disponibilidad	0,99
Reflectividad	0,935
Limpieza	0,98

En la Fig.3.8 puede verse una captura de pantalla de la pestaña de configuración del tipo de SCA en SAM. En la Fig.3.9, puede verse la pestaña correspondiente a la configuración de HCE. En este caso el modelo empleado es el UVAC 3, de Solel, con los parámetros que se indican en la tabla 3.4.

IPH Trough, No financial Location and Resource System Design Solar Field Collectors (SCAs) Receivers (HCEs) Thermal Storage System Control	Design Point Parameters <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">-Solar Field</td> <td style="width: 50%;">-Heat Sink</td> </tr> <tr> <td>Design point DNI</td> <td>2520 W/m²</td> </tr> <tr> <td>Target solar multiple</td> <td>2.51</td> </tr> <tr> <td>Target receiver thermal power</td> <td>750.00 MWt</td> </tr> <tr> <td>Loop inlet HTF temperature</td> <td>293 °C</td> </tr> <tr> <td>Loop outlet HTF temperature</td> <td>393 °C</td> </tr> </table> System Summary <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">Actual number of loops</td> <td style="width: 50%;">120</td> </tr> <tr> <td>Total aperture reflective area</td> <td>392,400.0 m²</td> </tr> <tr> <td style="border-top: none;">Actual solar multiple</td> <td style="border-top: none;">2.51</td> </tr> <tr> <td style="border-top: none;">Actual field thermal output</td> <td style="border-top: none;">752.41 MWt</td> </tr> </table>	-Solar Field	-Heat Sink	Design point DNI	2520 W/m ²	Target solar multiple	2.51	Target receiver thermal power	750.00 MWt	Loop inlet HTF temperature	293 °C	Loop outlet HTF temperature	393 °C	Actual number of loops	120	Total aperture reflective area	392,400.0 m ²	Actual solar multiple	2.51	Actual field thermal output	752.41 MWt
-Solar Field	-Heat Sink																				
Design point DNI	2520 W/m ²																				
Target solar multiple	2.51																				
Target receiver thermal power	750.00 MWt																				
Loop inlet HTF temperature	293 °C																				
Loop outlet HTF temperature	393 °C																				
Actual number of loops	120																				
Total aperture reflective area	392,400.0 m ²																				
Actual solar multiple	2.51																				
Actual field thermal output	752.41 MWt																				

Fig. 3.8. Configuración SAM. Configuración del SCA SenerTrough I a partir del modelo EuroTrough ET150

IPH Trough, No financial Location and Resource Solar Field Collectors (SCAs) Receivers (HCEs) Thermal Storage System Control	Solar Field Design Point <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">Single loop aperture</td> <td style="width: 50%;">3,270 m²</td> </tr> <tr> <td>Loop optical efficiency</td> <td>0.772</td> </tr> <tr> <td>Total loop conversion efficiency</td> <td>0.761</td> </tr> <tr> <td>Total required aperture, SM=1</td> <td>156,456 m²</td> </tr> <tr> <td>Required number of loops, SM=1</td> <td>48</td> </tr> <tr> <td>Total tracking power</td> <td>60,000 W</td> </tr> </table> Solar Field Parameters <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">Row spacing</td> <td style="width: 50%;">16.25 m</td> </tr> <tr> <td>Header pipe roughness</td> <td>4.57e-05 m</td> </tr> <tr> <td>HTF pump efficiency</td> <td>0.85</td> </tr> <tr> <td>Piping thermal loss coefficient</td> <td>0 W/m²-K</td> </tr> <tr> <td>Wind stow speed</td> <td>25 m/s</td> </tr> <tr> <td>Receiver startup delay time</td> <td>0 hr</td> </tr> <tr> <td>Receiver startup delay energy fraction</td> <td>0 -</td> </tr> <tr> <td>Collector startup energy</td> <td>0 kWhe/sca</td> </tr> <tr> <td>Tracking power per SCA</td> <td>125 W/sca</td> </tr> <tr> <td>Number of field subsections</td> <td>1</td> </tr> <tr> <td>Allow partial defocusing</td> <td>Simultaneous <input checked="" type="checkbox"/></td> </tr> </table> Heat Transfer Fluid <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">Field HTF fluid</td> <td style="width: 50%;">Therminol VP-1 <input type="button" value="Edit..."/></td> </tr> <tr> <td>Field HTF min operating temp</td> <td>12 °C</td> </tr> <tr> <td>Field HTF max operating temp</td> <td>400 °C</td> </tr> <tr> <td>Freeze protection temp</td> <td>50 °C</td> </tr> <tr> <td>Min single loop flow rate</td> <td>1.7 kg/s</td> </tr> <tr> <td>Max single loop flow rate</td> <td>20 kg/s</td> </tr> <tr> <td>Min field flow velocity</td> <td>0.6 m/s</td> </tr> <tr> <td>Max field flow velocity</td> <td>8.3 m/s</td> </tr> <tr> <td>Cold Headers</td> <td><input type="checkbox"/></td> </tr> <tr> <td>Hot Headers</td> <td><input type="checkbox"/></td> </tr> <tr> <td>Header design min flow velocity</td> <td>2 m/s</td> </tr> <tr> <td>Header design max flow velocity</td> <td>10 m/s</td> </tr> </table> Collector Orientation <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">Collector tilt</td> <td style="width: 50%;">0 deg</td> </tr> <tr> <td>Collector azimuth</td> <td>0 deg</td> </tr> <tr> <td>Tilt: horizontal=0, vertical=90</td> <td>Azimuth: equator=0, west=90</td> </tr> <tr> <td>Stow angle</td> <td>170 deg</td> </tr> <tr> <td>Deploy angle</td> <td>10 deg</td> </tr> </table> Mirror Washing <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">Water usage per wash</td> <td style="width: 50%;">0.7 L/m², aper.</td> </tr> <tr> <td>Washes per year</td> <td>12</td> </tr> </table> Plant Heat Capacity <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">Hot piping thermal inertia</td> <td style="width: 50%;">0 kWh/K-MWt</td> </tr> <tr> <td>Cold piping thermal inertia</td> <td>0 kWh/K-MWt</td> </tr> <tr> <td>Field loop piping thermal inertia</td> <td>0 Wh/K-m</td> </tr> </table> Land Area <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">Solar field area</td> <td style="width: 50%;">273 acres</td> </tr> <tr> <td>Non-solar field land area multiplier</td> <td>1.1</td> </tr> <tr> <td>Total land area</td> <td>300 acres</td> </tr> </table> Single Loop Configuration <p>The specification below is only for one loop in the solar field. Usage tip: To configure the loop, choose whether to edit SCAs, HCEs or defocus order. Select assemblies by clicking one or dragging the mouse over multiple items. Assign types to selected items by pressing keys 1-4.</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">Number of SCA/HCE assemblies per loop:</td> <td style="width: 50%;">4 <input type="radio"/> Edit SCAs <input type="radio"/> Edit HCEs <input type="radio"/> Edit Defocus Order <input type="button" value="Reset Defocus"/></td> </tr> <tr> <td colspan="2"> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 25%;">SCA: 1</td> <td style="width: 25%;">SCA: 1</td> <td style="width: 25%;">SCA: 1</td> <td style="width: 25%;">SCA: 1</td> </tr> <tr> <td>HCE: 1 DF# 4</td> <td>HCE: 1 DF# 3</td> <td>HCE: 1 DF# 2</td> <td>HCE: 1 DF# 1</td> </tr> </table> </td> </tr> </table>	Single loop aperture	3,270 m ²	Loop optical efficiency	0.772	Total loop conversion efficiency	0.761	Total required aperture, SM=1	156,456 m ²	Required number of loops, SM=1	48	Total tracking power	60,000 W	Row spacing	16.25 m	Header pipe roughness	4.57e-05 m	HTF pump efficiency	0.85	Piping thermal loss coefficient	0 W/m ² -K	Wind stow speed	25 m/s	Receiver startup delay time	0 hr	Receiver startup delay energy fraction	0 -	Collector startup energy	0 kWhe/sca	Tracking power per SCA	125 W/sca	Number of field subsections	1	Allow partial defocusing	Simultaneous <input checked="" type="checkbox"/>	Field HTF fluid	Therminol VP-1 <input type="button" value="Edit..."/>	Field HTF min operating temp	12 °C	Field HTF max operating temp	400 °C	Freeze protection temp	50 °C	Min single loop flow rate	1.7 kg/s	Max single loop flow rate	20 kg/s	Min field flow velocity	0.6 m/s	Max field flow velocity	8.3 m/s	Cold Headers	<input type="checkbox"/>	Hot Headers	<input type="checkbox"/>	Header design min flow velocity	2 m/s	Header design max flow velocity	10 m/s	Collector tilt	0 deg	Collector azimuth	0 deg	Tilt: horizontal=0, vertical=90	Azimuth: equator=0, west=90	Stow angle	170 deg	Deploy angle	10 deg	Water usage per wash	0.7 L/m ² , aper.	Washes per year	12	Hot piping thermal inertia	0 kWh/K-MWt	Cold piping thermal inertia	0 kWh/K-MWt	Field loop piping thermal inertia	0 Wh/K-m	Solar field area	273 acres	Non-solar field land area multiplier	1.1	Total land area	300 acres	Number of SCA/HCE assemblies per loop:	4 <input type="radio"/> Edit SCAs <input type="radio"/> Edit HCEs <input type="radio"/> Edit Defocus Order <input type="button" value="Reset Defocus"/>	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 25%;">SCA: 1</td> <td style="width: 25%;">SCA: 1</td> <td style="width: 25%;">SCA: 1</td> <td style="width: 25%;">SCA: 1</td> </tr> <tr> <td>HCE: 1 DF# 4</td> <td>HCE: 1 DF# 3</td> <td>HCE: 1 DF# 2</td> <td>HCE: 1 DF# 1</td> </tr> </table>		SCA: 1	SCA: 1	SCA: 1	SCA: 1	HCE: 1 DF# 4	HCE: 1 DF# 3	HCE: 1 DF# 2	HCE: 1 DF# 1
Single loop aperture	3,270 m ²																																																																																																
Loop optical efficiency	0.772																																																																																																
Total loop conversion efficiency	0.761																																																																																																
Total required aperture, SM=1	156,456 m ²																																																																																																
Required number of loops, SM=1	48																																																																																																
Total tracking power	60,000 W																																																																																																
Row spacing	16.25 m																																																																																																
Header pipe roughness	4.57e-05 m																																																																																																
HTF pump efficiency	0.85																																																																																																
Piping thermal loss coefficient	0 W/m ² -K																																																																																																
Wind stow speed	25 m/s																																																																																																
Receiver startup delay time	0 hr																																																																																																
Receiver startup delay energy fraction	0 -																																																																																																
Collector startup energy	0 kWhe/sca																																																																																																
Tracking power per SCA	125 W/sca																																																																																																
Number of field subsections	1																																																																																																
Allow partial defocusing	Simultaneous <input checked="" type="checkbox"/>																																																																																																
Field HTF fluid	Therminol VP-1 <input type="button" value="Edit..."/>																																																																																																
Field HTF min operating temp	12 °C																																																																																																
Field HTF max operating temp	400 °C																																																																																																
Freeze protection temp	50 °C																																																																																																
Min single loop flow rate	1.7 kg/s																																																																																																
Max single loop flow rate	20 kg/s																																																																																																
Min field flow velocity	0.6 m/s																																																																																																
Max field flow velocity	8.3 m/s																																																																																																
Cold Headers	<input type="checkbox"/>																																																																																																
Hot Headers	<input type="checkbox"/>																																																																																																
Header design min flow velocity	2 m/s																																																																																																
Header design max flow velocity	10 m/s																																																																																																
Collector tilt	0 deg																																																																																																
Collector azimuth	0 deg																																																																																																
Tilt: horizontal=0, vertical=90	Azimuth: equator=0, west=90																																																																																																
Stow angle	170 deg																																																																																																
Deploy angle	10 deg																																																																																																
Water usage per wash	0.7 L/m ² , aper.																																																																																																
Washes per year	12																																																																																																
Hot piping thermal inertia	0 kWh/K-MWt																																																																																																
Cold piping thermal inertia	0 kWh/K-MWt																																																																																																
Field loop piping thermal inertia	0 Wh/K-m																																																																																																
Solar field area	273 acres																																																																																																
Non-solar field land area multiplier	1.1																																																																																																
Total land area	300 acres																																																																																																
Number of SCA/HCE assemblies per loop:	4 <input type="radio"/> Edit SCAs <input type="radio"/> Edit HCEs <input type="radio"/> Edit Defocus Order <input type="button" value="Reset Defocus"/>																																																																																																
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 25%;">SCA: 1</td> <td style="width: 25%;">SCA: 1</td> <td style="width: 25%;">SCA: 1</td> <td style="width: 25%;">SCA: 1</td> </tr> <tr> <td>HCE: 1 DF# 4</td> <td>HCE: 1 DF# 3</td> <td>HCE: 1 DF# 2</td> <td>HCE: 1 DF# 1</td> </tr> </table>		SCA: 1	SCA: 1	SCA: 1	SCA: 1	HCE: 1 DF# 4	HCE: 1 DF# 3	HCE: 1 DF# 2	HCE: 1 DF# 1																																																																																								
SCA: 1	SCA: 1	SCA: 1	SCA: 1																																																																																														
HCE: 1 DF# 4	HCE: 1 DF# 3	HCE: 1 DF# 2	HCE: 1 DF# 1																																																																																														

Fig. 3.9. Configuración SAM. Configuración del HCE UVAC 3 con vacío

TABLA 3.4. Configuración del HCE modelo UVAC 3 de Solel

Parámetro	Valor
Longitud	4,05 (m)
Factor de longitud efectiva, γ_L	0,96

Tabla 3.4 continúa desde la página anterior

Parámetro	Valor
Factor de interceptación geométrica, γ_g	0,96
Diámetro interior del tubo absorbedor, D_{ri}	0,066 (m)
Diámetro exterior del tubo absorbedor, D_{ro}	0,070 (m)
Diámetro interior de la envolvente de vidrio, D_{gi}	0,115 (m)
Diámetro exterior de la envolvente de vidrio, D_{go}	0,121 (m)
Rugosidad interior	0,000045 (m)
Factor de emisividad, A_1	0,000206
Factor de emisividad, A_0	0,0430
Absortibilidad solar del receptor	0,96
Transmisividad del vidrio	0,96
Valor mínimo del número de Reynolds recomendado	2300
Distancia de separación entre brazos de soportación	4,05 (m)

Se ha considerado que todos los HCE del campo se encuentran en buen estado de vacío.

3.4.2. Guía para la configuración de la simulación en Python

Para la configuración de la simulación con nuestro código se debe crear un archivo en formato *JSON* con el que se le pasa toda la información necesaria. Con el fin de facilitar la creación de este archivo de configuración se ha desarrollado una sencilla interfaz, *interface.py*, que sirve de guía durante el proceso. Aprovecharemos este apartado para crear la configuración al tiempo que nos sirve de breve guía de uso de la interfaz de usuario.

Configuración de la simulación

En la Fig.3.10 se muestra la primera pantalla de la interfaz de configuración. Al abrir por primera vez se carga una plantilla con los datos de una configuración preestablecida. En las siguientes figuras se muestran los datos ya modificados según nuestras necesidades. El archivo de origen de datos que hemos seleccionado incluye los mismos datos meteoro-

lógicos que se han empleado para SAM y, además, los datos de temperatura de entrada del HTF al campo solar que ha generado SAM en la simulación IPH. También emplearemos el caudal calculado por SAM para compararlo con el que calcula nuestro simulador.

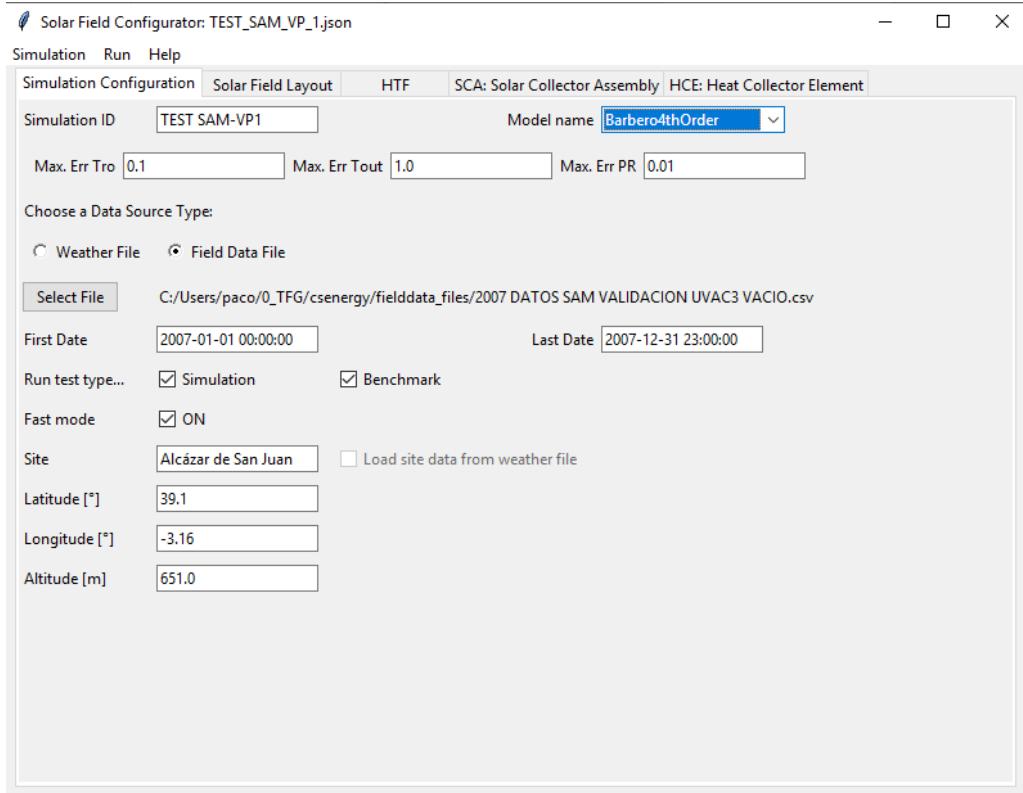


Fig. 3.10. Configuración de la simulación. Selección del tipo de simulación, fichero de datos y lugar de emplazamiento.

En esta primera pestaña de la interfaz, el usuario puede dar un nombre identificador a la simulación. El campo *Model name* es desplegable y permite seleccionar qué modelo se empleará para el cálculo del rendimiento térmico. Seguidamente, hay cuatro campos que permiten definir el máximo error permitido para la finalización del proceso iterativo:

- *Max. Err Tro* es el error absoluto en la temperatura de pared exterior, T_{ro}
- *Max. Err Tout* es el error absoluto en la temperatura de salida del lazo, T_{out}
- *Max. Err PR* es el error absoluto en el rendimiento térmico, η

De esta forma, el proceso iterativo continúa mientras los valores calculados entre dos iteraciones sucesivas presenten una diferencia superior a estos valores. Internamente se

ha fijado un máximo de 1000 iteraciones, de tal forma que si no se alcanza convergencia tras ese número de ciclos el proceso muestra un mensaje y salta al siguiente punto.

Según se seleccione un origen de datos que incluya también datos de caudal y temperaturas de entrada y salida o, por el contrario, puramente meteorológico (*Field Data File* o *Weather File*), el programa nos permitirá, o no, realizar una simulación tipo *benchmark*.

La casilla de selección *Fastmode* permite indicar que se realice la simulación considerando todos los lazos iguales o no. Esta opción es la única empleada en este trabajo pues, aunque es posible modificar a mano la estructura del campo solar para configurar cada lazo independientemente, en realidad está pensada para la inclusión en un futuro de un módulo que permita leer el estado de cada lazo en cada momento.

Si se selecciona la casilla *Load site data from weather file*, la interfaz carga los datos del emplazamiento del fichero tipo *Weather File* durante el proceso de carga. En caso contrario, los debe introducir el usuario.

Configuración del Campo Solar

En la Fig.3.11 vemos cómo se configura el campo solar en un proceso similar al de SAM. Para esta simulación, con el fin acelerar el proceso de simulación, consideraremos que hay 2 HCEs en cada SCA. Es decir, cada HCE tiene una longitud de 72,9 m, que está por debajo del límite de 100 m que nos hemos impuesto como criterio de validez del tamaño de malla para el modelo unidimensional. El caudal mínimo de recirculación en cada lazo es de 1,7 kg/s. La temperatura de salida deseada es de 393°C (666,15 K).

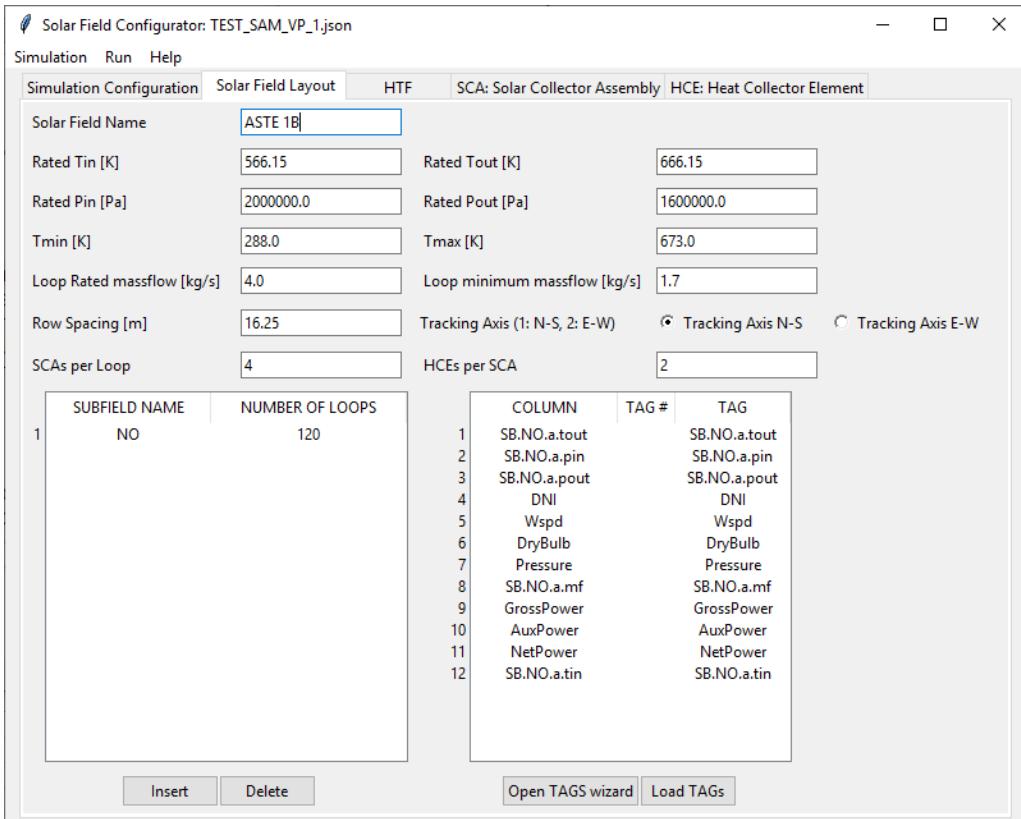


Fig. 3.11. Configuración de la simulación. Configuración del campo solar, número de subcampos, lazos, configuración de los lazos, valores nominales y asistente para relacionar los identificadores de las columnas del archivo de origen de datos con los que maneja el programa.

Además de los valores nominales, que son empleados como punto de partida en las simulaciones, vemos que podemos configurar la orientación del eje del seguidor y el espaciado entre lazos para el cálculo de sombras. En la configuración de esta simulación hemos supuesto que existe un único subcampo con 120 lazos. La aplicación permitiría añadir varios subcampos mediante una lista de pares *nombre de subcampo - número de lazos*. El nombre que se dé a los subcampos es relevante en el sentido de que, posteriormente, los archivos CSV que se generarán utilizarán una nomenclatura basada en esos nombres para mostrar las variables de temperatura, caudal, potencia, rendimiento, etc., de cada subcampo.

En la Fig.3.12 se muestra el proceso que permite establecer una relación entre los tags o nombres que reciben los valores en el archivo de datos de campo y los valores que empleará el programa para el informe. Este paso solo es necesario cuando vamos

a utilizar, como origen de datos, un archivo con columnas cuyos encabezados tienen un descriptor diferente al que el programa va a emplear. Una vez establecida esta relación entre identificadores y guardada en el archivo de configuración, ya no es necesario realizar este paso en las siguientes configuraciones que empleen el mismo archivo de origen de datos. El asistente muestra una lista (Fig.3.13) con los *tags* disponibles junto a un número. El usuario debe indicar qué número corresponde a cada variable. Por ejemplo, el caudal másico del subcampo NO viene dado por el identificador número 10, *1310-FX-1809*. Cuando ha terminado de asignar los números, debe pulsar *Load TAGs* para que se cargue la lista, tal y como se ve en la Fig.3.14. Al quedar asociados de este modo, cuando el simulador cargue los datos del campo sabrá que debe emplear los datos de la columna con encabezado *1310-FX-1809* como datos de caudal de ese subcampo.

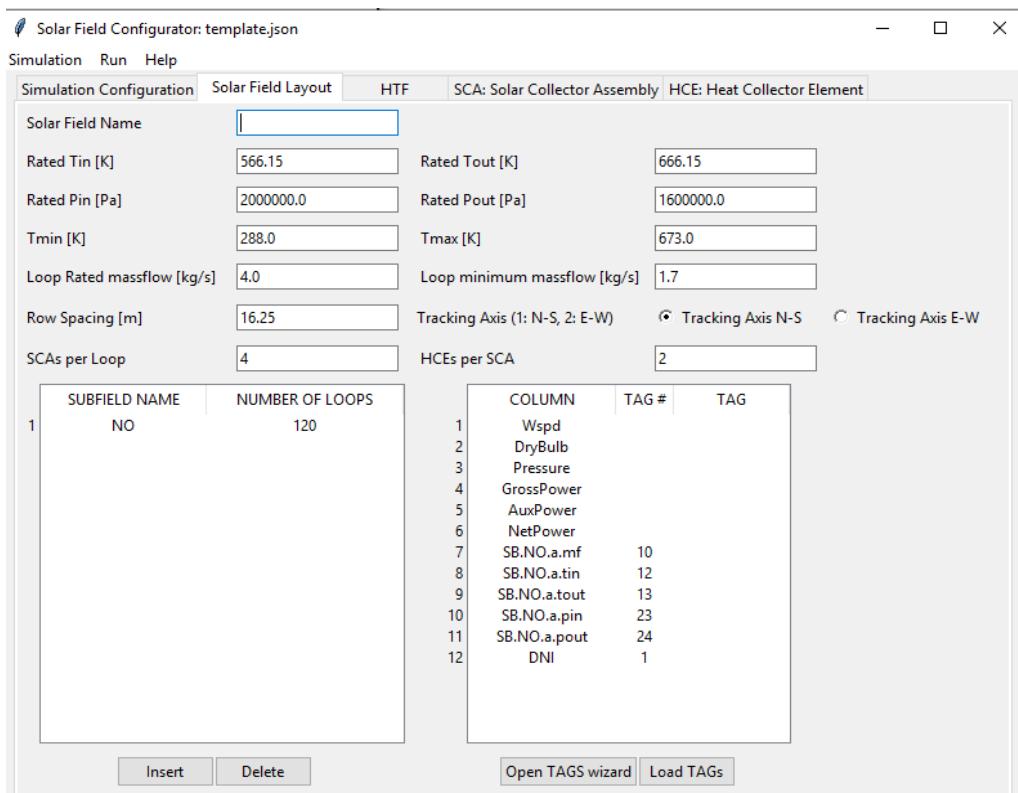


Fig. 3.12. Proceso de asignación de *tags* a variables de entrada/salida

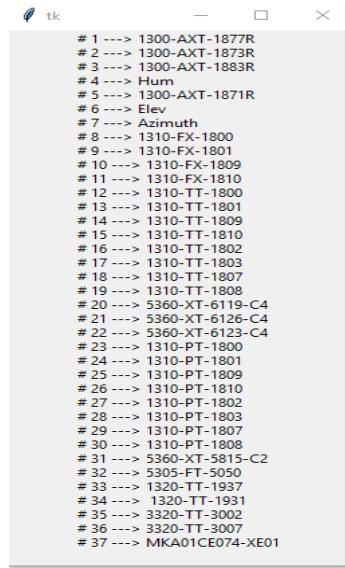


Fig. 3.13. Ventana con la lista de tags disponibles

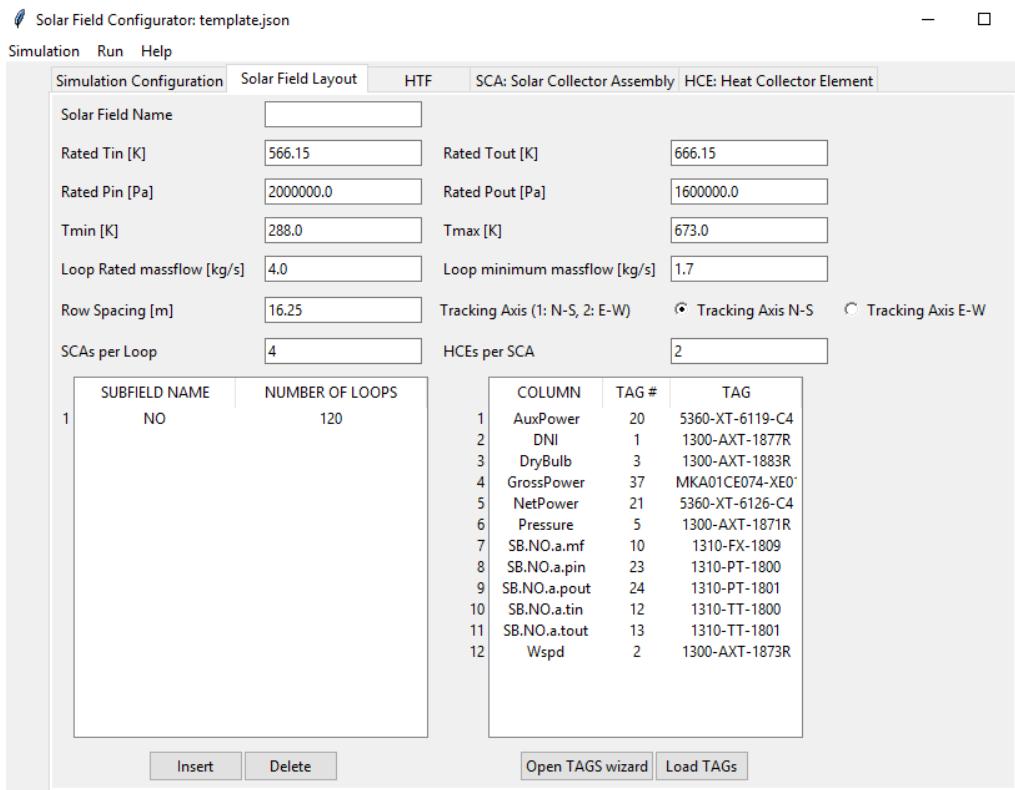


Fig. 3.14. Lista de asignaciones tag-variable completa

Selección del fluido caloportador

El fluido caloportador *Therminol VP-1* se configura tal y como puede verse en la Fig.3.15. El usuario puede optar por seleccionar los fluidos disponibles en la librería Co-

olProp, con lo que los valores máximos y mínimos de trabajo quedan fijados a los que establece dicha librería. Actualmente la interfaz permite la selección de dos fluidos desde esta librería: *Terminol VP-1* y *Syltherm 800*. Otra posibilidad es seleccionar alguno de los fluidos para los que se dispone de los coeficientes de los polinomios que definen sus propiedades. En este caso, el fichero con la lista de fluidos se denomina *fluids_lib.json* y se puede editar directamente para añadir nuevos coeficientes. Una vez el fichero se ha cargado, se selecciona en el campo desplegable el fluido que se desee y los coeficientes se cargan en la tabla donde también pueden editarse.

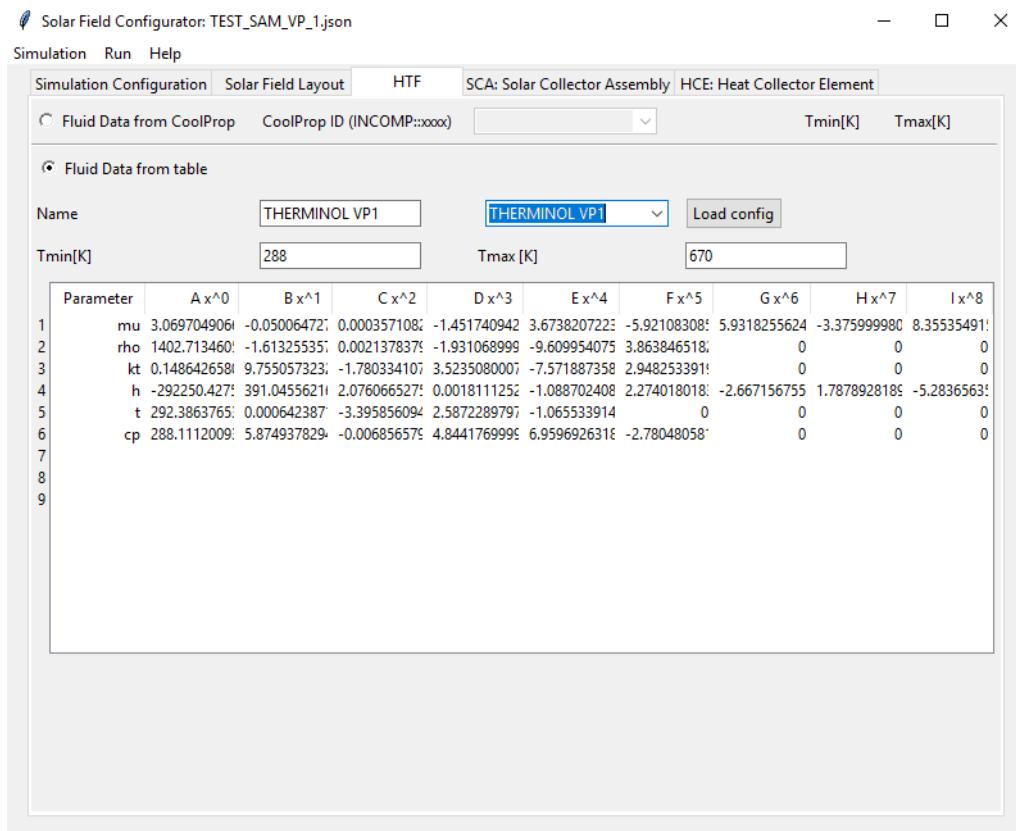


Fig. 3.15. Configuración de la simulación. Selección y configuración del fluido caloportador

Configuración del modelo de SCA

En la pestaña de configuración de SCA también disponemos de un campo desplegable en el que seleccionar algún modelo de SCA previamente almacenado en un fichero, en este caso denominado *SCA_lib.json*. De esta forma disponemos de los datos de distancia focal, los factores para el cálculo del IAM dados por el fabricante o la apertura. Todos los

datos se pueden editar y modificar para realizar diversas pruebas, como variar las factores de limpieza, reflectividad, etc.

La pestaña de configuración de SCA con los datos correspondientes al modelo Sener-Trough ya cargados puede verse en la Fig.3.16.

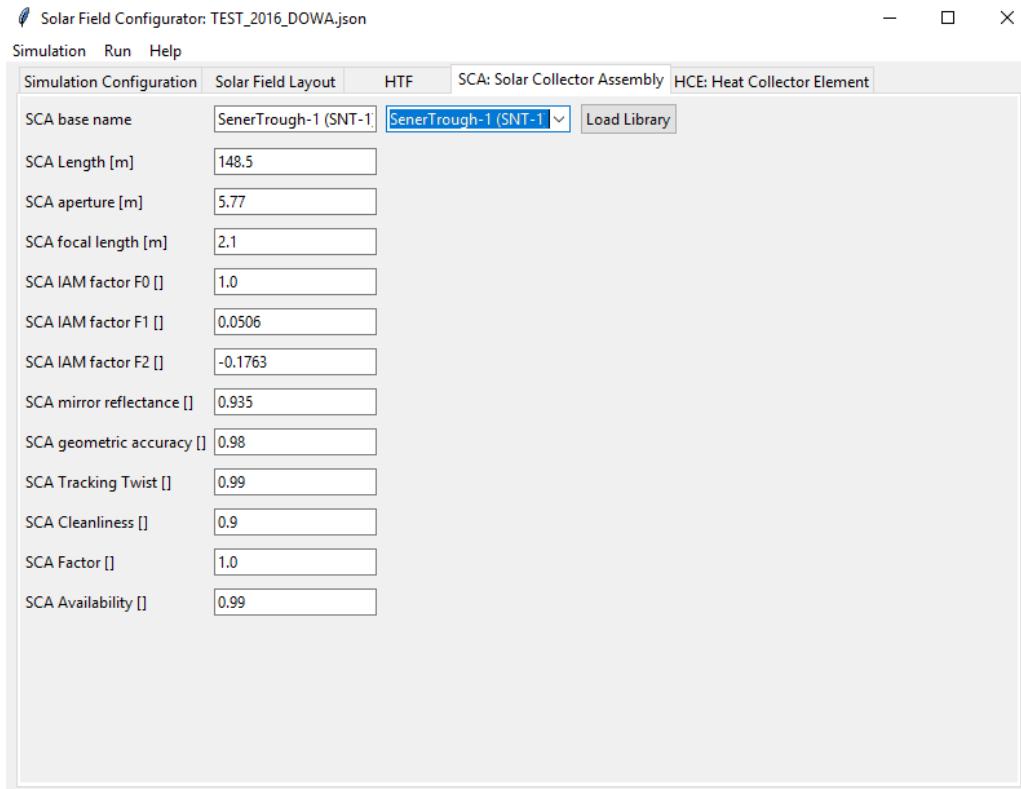


Fig. 3.16. Configuración de la simulación. Selección del modelo de SCA y configuración

Configuración del modelo de HCE

Finalmente, en la Fig.3.17 puede verse la configuración del modelo de HCE. Nuevamente disponemos de un archivo que almacena los datos de algunos modelos, denominado en este caso *HCE_lib.json*. Una vez seleccionado el modelo que queremos, los datos se cargan y pueden ser editados. Es importante destacar que la longitud el HCE debe modificarse teniendo en cuenta el tamaño del SCA y el número de HCEs por SCA que hemos preconfigurado. Un pequeño texto nos da información sobre el numero máximo de HCEs que caben en el SCA según la longitud que indiquemos.

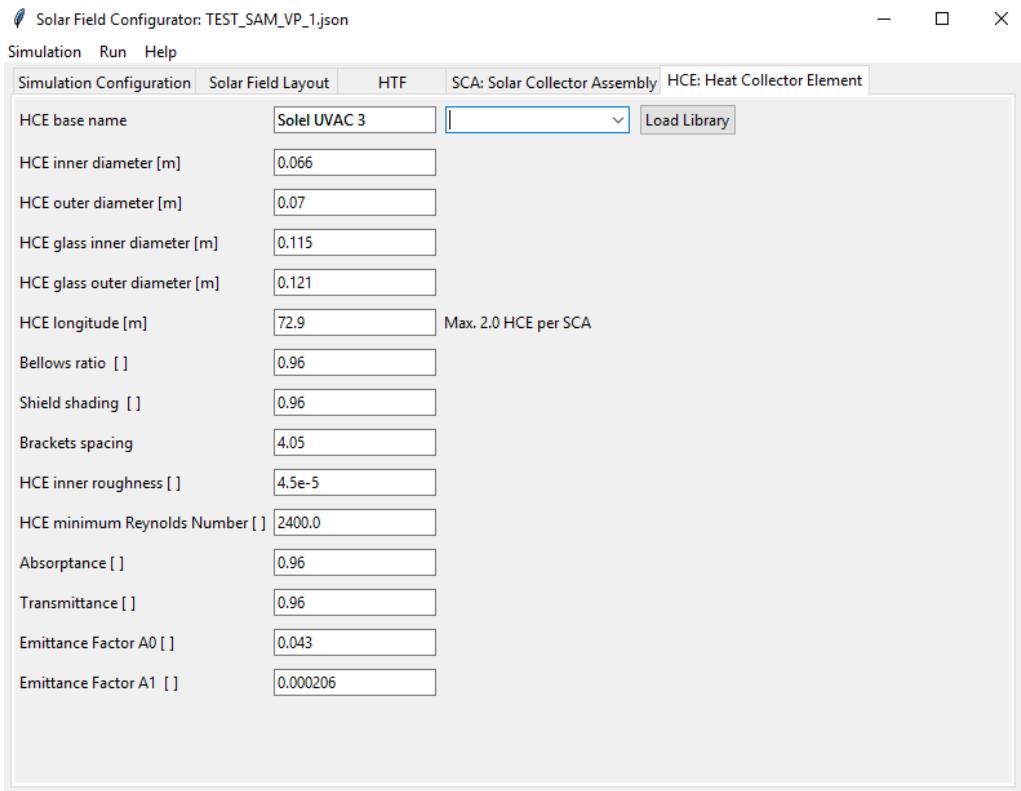


Fig. 3.17. Configuración de la simulación. Selección del modelo de HCE y configuración

Lanzamiento de la simulación

Una vez hemos terminado de configurar nuestra simulación debemos guardarla con el nombre que queramos en un fichero. El resultado puede verse en el código 3.1. En este caso, los *tags* coinciden con los nombres de las variables porque ya se habían modificado previamente los encabezados de las columnas del archivo de origen de datos.

CÓDIGO 3.1. Fichero de configuración de la simulación en formato JSON

```

1  {
2      "simulation": {
3          "ID": "TEST 2007 IPH",
4          "datatype": 2,
5          "simulation": true,
6          "benchmark": true,
7          "fastmode": true,
8          "filename": "2007 IPH LOOP TEMP.csv",
9          "filepath": "C:/Users/paco/0_TFG/csenergy/fielddata_files/",
10         "first_date": "2007/01/31 00:00",

```

```

11 "last_date": "2007/12/31 23:00"},  

12 "model": {  

13     "name": "Barbero4thOrder",  

14     "max_err_t": 0.1,  

15     "max_err_tro": 0.1,  

16     "max_err_pr": 0.01},  

17 "site": {  

18     "name": "Alcazar de San Juan",  

19     "latitude": 39.1,  

20     "longitude": -3.16,  

21     "altitude": 651.0},  

22 "loop": {  

23     "scas": 4,  

24     "hces": 2,  

25     "rated_tin": 566.15,  

26     "rated_tout": 666.15,  

27     "rated_pin": 2000000.0,  

28     "rated_pout": 1600000.0,  

29     "tmin": 288.0,  

30     "tmax": 673.0,  

31     "rated_massflow": 4.0,  

32     "min_massflow": 1.7,  

33     "row_spacing": 16.25,  

34     "Tracking Type": 1},  

35 "SCA": {  

36     "Name": "SenerTrough-1 (SNT-1)",  

37     "SCA Length": 148.5,  

38     "Aperture": 5.77,  

39     "Focal Len": 2.1,  

40     "IAM Coefficient F0": 1.0,  

41     "IAM Coefficient F1": 0.0506,  

42     "IAM Coefficient F2": -0.1763,  

43     "Track Twist": 0.99,  

44     "Geom.Accuracy": 0.98,  

45     "Reflectance": 0.935,  

46     "Cleanliness": 0.98,  

47     "Factor": 1.0,  

48     "Availability": 0.99},  

49 "HCE": {
```

```

50     "Name": "Solel UVAC 3",
51     "Length": 72.9,
52     "Bellows ratio": 0.96,
53     "Shield shading": 1.0,
54     "Absorber tube inner diameter": 0.066,
55     "Absorber tube outer diameter": 0.07,
56     "Glass envelope inner diameter": 0.115,
57     "Glass envelope outer diameter": 0.121,
58     "Min Reynolds": 2400.0,
59     "Inner surface roughness": 4.5e-05,
60     "Envelope transmittance": 0.965,
61     "Absorber emittance factor A0": 0.043,
62     "Absorber emittance factor A1": 0.000206,
63     "Absorber absorptance": 0.96,
64     "Brackets": 4.05
65   },
66   "HTF": {
67     "source": "table",
68     "name": "THERMINOL VP1",
69     "mu": [ 1.487, -0.02186, 0.0001400, -5.092e-07, 1.148e-09, -1.642e
-12, 1.454e-15, -7.286e-19, 1.582e-22],
70     "rho": [1403, -1.614, 0.002138, -1.931e-06, -9.610e-21, 3.864e-24,
0, 0, 0],
71     "kt": [0.1486, 9.755e-06, -1.780e-07, 3.523e-12, -7.572e-25, 2.948
e-28, 0, 0, 0],
72     "h": [-292250, 391.0, 2.076, 0.001811, -1.089e-05, 2.274e-08,
-2.667e-11, 1.788e-14, -5.284e-18],
73     "t": [292.4, 0.0006424, -3.396e-10, 2.587e-16, -1.065e-22, 0, 0,
0, 0],
74     "cp": [288.1, 5.875, -0.006857, 4.844e-06, 6.960e-20, -2.780e-23,
0, 0, 0],
75     "tmax": 673.0,
76     "tmin": 288.0},
77   "subfields": [{"name": "NO", "loops": 120}],
78   "tags": {
79     "SB.NO.a.tin": "SB.NO.a.tin",
80     "SB.NO.a.tout": "SB.NO.a.tout",
81     "SB.NO.a.pin": "SB.NO.a.pin",
82     "SB.NO.a.pout": "SB.NO.a.pout",

```

```

83     "DNI": "DNI",
84     "Wspd": "Wspd",
85     "DryBulb": "DryBulb",
86     "Pressure": "Pressure",
87     "SB.NO.a.mf": "SB.NO.a.mf",
88     "GrossPower": "GrossPower",
89     "AuxPower": "AuxPower",
90     "NetPower": "NetPower"
91   }

```

Desde la propia interfaz puede lanzarse la simulación mediante la opción de menú *Run*, aunque también se puede lanzar la ejecución mediante un *script* como el que se muestra en el código 3.2, gracias a que se ha creado la Clase **SolarFieldSimulation** que se encarga de recibir el archivo de configuración y lanzar la simulación conforme a lo que en él se especifique.

CÓDIGO 3.2. Script para lanzar la simulación a partir del fichero JSON

```

1 import json
2 import csenergy as cs
3 with open("./saved_configurations/TEST_2007_IPH_VP1.json") as
4     simulation_file:
5         SIMULATION_SETTINGS = json.load(simulation_file)
6 SIM = cs.SolarFieldSimulation(SIMULATION_SETTINGS)
7 SIM.runSimulation()

```

En la Fig.3.18 vemos una captura de pantalla del inicio de la simulación en la consola de Python. Cuando el programa finaliza la ejecución, se guarda un fichero en formato CSV con todos los datos calculados. El tamaño de este fichero puede ser de varios MB en el caso en que no se haya seleccionado la opción *Fastmode*, ya que de este modo se están almacenando, para cada fecha de cálculo, los datos de temperatura, caudal, potencia, y rendimiento de cada lazo, los agregados de los subcampos y el agregado del campo completo. Es decir, que puede tratarse de una tabla con varios cientos de columnas y, en caso de que la simulación sea para un año completo con datos horarios, de 8760 registros o filas.

```

In [1]: runfile('C:/Users/paco/0_TFG/csenergy/interface.py', wdir='C:/Users/paco/0_TFG/csenergy', post_mortem=True)
Running simulation for source data file: 2007 IPH LOOP TEMP.csv from: 2007/01/01 00:00 to 2007/12/31 23:00
Model: BarberoSimplified
Simulation: True ; Benchmark: True ; FastMode: True
Site: Alcázar de San Juan @ Lat: 39.10°, Long: -3.16°, Alt: 651.0 m
Loops: 120 SCA/loop: 4 HCE/SCA: 2
SCA model: SenerTrough-1 (SNT-1)
HCE model: Solel UVAC 3
HTF form table: THERMINOL VP1
-----
SIMULATION: 2007-01-01 00:00:00 DNI: 0 W/m2 Qm: 204.0kg/s Tin: 170.7° Tout: 165.3°C
BENCHMARK: 2007-01-01 00:00:00 DNI: 0 W/m2 act_Qm: 204.0kg/s act_Tin: 170.7° act_Tout: 167.1° Tout: 165.3°
SIMULATION: 2007-01-01 01:00:00 DNI: 0 W/m2 Qm: 204.0kg/s Tin: 169.6° Tout: 164.3°C
BENCHMARK: 2007-01-01 01:00:00 DNI: 0 W/m2 act_Qm: 204.0kg/s act_Tin: 169.6° act_Tout: 166.1° Tout: 164.3°
SIMULATION: 2007-01-01 02:00:00 DNI: 0 W/m2 Qm: 204.0kg/s Tin: 168.5° Tout: 163.2°C
BENCHMARK: 2007-01-01 02:00:00 DNI: 0 W/m2 act_Qm: 204.0kg/s act_Tin: 168.5° act_Tout: 165.1° Tout: 163.2°
SIMULATION: 2007-01-01 03:00:00 DNI: 0 W/m2 Qm: 204.0kg/s Tin: 167.3° Tout: 162.2°C
BENCHMARK: 2007-01-01 03:00:00 DNI: 0 W/m2 act_Qm: 204.0kg/s act_Tin: 167.3° act_Tout: 164.0° Tout: 162.2°
SIMULATION: 2007-01-01 04:00:00 DNI: 0 W/m2 Qm: 204.0kg/s Tin: 166.2° Tout: 161.1°C
BENCHMARK: 2007-01-01 04:00:00 DNI: 0 W/m2 act_Qm: 204.0kg/s act_Tin: 166.2° act_Tout: 162.9° Tout: 161.1°
SIMULATION: 2007-01-01 05:00:00 DNI: 0 W/m2 Qm: 204.0kg/s Tin: 165.1° Tout: 160.1°C
BENCHMARK: 2007-01-01 05:00:00 DNI: 0 W/m2 act_Qm: 204.0kg/s act_Tin: 165.1° act_Tout: 161.8° Tout: 160.1°
SIMULATION: 2007-01-01 06:00:00 DNI: 0 W/m2 Qm: 204.0kg/s Tin: 164.0° Tout: 159.0°C
BENCHMARK: 2007-01-01 06:00:00 DNI: 0 W/m2 act_Qm: 204.0kg/s act_Tin: 164.0° act_Tout: 160.8° Tout: 159.0°
SIMULATION: 2007-01-01 07:00:00 DNI: 0 W/m2 Qm: 204.0kg/s Tin: 163.0° Tout: 158.0°C
BENCHMARK: 2007-01-01 07:00:00 DNI: 0 W/m2 act_Qm: 204.0kg/s act_Tin: 163.0° act_Tout: 159.8° Tout: 158.0°
SIMULATION: 2007-01-01 08:00:00 DNI: 0 W/m2 Qm: 204.0kg/s Tin: 161.9° Tout: 157.0°C
BENCHMARK: 2007-01-01 08:00:00 DNI: 0 W/m2 act_Qm: 204.0kg/s act_Tin: 161.9° act_Tout: 158.8° Tout: 157.0°
SIMULATION: 2007-01-01 09:00:00 DNI: 200 W/m2 Qm: 204.0kg/s Tin: 163.1° Tout: 242.0°C
BENCHMARK: 2007-01-01 09:00:00 DNI: 200 W/m2 act_Qm: 204.0kg/s act_Tin: 163.1° act_Tout: 226.0° Tout: 242.0°
SIMULATION: 2007-01-01 10:00:00 DNI: 18 W/m2 Qm: 204.0kg/s Tin: 175.6° Tout: 176.1°C
BENCHMARK: 2007-01-01 10:00:00 DNI: 18 W/m2 act_Qm: 204.0kg/s act_Tin: 175.6° act_Tout: 188.2° Tout: 176.1°
SIMULATION: 2007-01-01 11:00:00 DNI: 30 W/m2 Qm: 204.0kg/s Tin: 183.3° Tout: 184.8°C
BENCHMARK: 2007-01-01 11:00:00 DNI: 30 W/m2 act_Qm: 204.0kg/s act_Tin: 183.3° act_Tout: 184.4° Tout: 184.8°
SIMULATION: 2007-01-01 12:00:00 DNI: 112 W/m2 Qm: 204.0kg/s Tin: 185.9° Tout: 204.8°C
BENCHMARK: 2007-01-01 12:00:00 DNI: 112 W/m2 act_Qm: 204.0kg/s act_Tin: 185.9° act_Tout: 202.0° Tout: 204.8°
SIMULATION: 2007-01-01 13:00:00 DNI: 223 W/m2 Qm: 204.0kg/s Tin: 191.3° Tout: 243.6°C
BENCHMARK: 2007-01-01 13:00:00 DNI: 223 W/m2 act_Qm: 204.0kg/s act_Tin: 191.3° act_Tout: 236.1° Tout: 243.6°
SIMULATION: 2007-01-01 14:00:00 DNI: 398 W/m2 Qm: 204.0kg/s Tin: 206.7° Tout: 329.4°C
BENCHMARK: 2007-01-01 14:00:00 DNI: 398 W/m2 act_Qm: 224.6kg/s act_Tin: 206.7° act_Tout: 304.6° Tout: 319.7°
-----
```

Fig. 3.18. Consola de Python mostrando el inicio de una simulación

Comentarios sobre los tiempos de ejecución

El tiempo empleado para la simulación de un día completo (24 registros) de un campo de 120 lazos con 4 SCAs por lazo y 2 HCEs por SCA (es decir, HCE de unos 72 m), con la opción *Fastmode* (todos los lazos idénticos) y en la que se ejecuta tanto la simulación tipo *simulation* como la *benchmark*, ronda los 4,75 segundos en un ordenador portátil con procesador Intel Core i5, 8 GB de memoria RAM y Windows 10 como sistema operativo. No parece un valor muy elevado pero si tenemos en cuenta los 365 días de un año el tiempo de ejecución llega a los 20 minutos aproximadamente para el año completo, muy superior a los tiempos habituales de simulación que emplean otras herramientas, como SAM, por ejemplo. Si vamos a una malla la mitad de pequeña (4 HCEs por SCA) el tiempo se duplica.

Según las pruebas realizadas, hemos podido comprobar que la convergencia se suele alcanzar al cabo de unas pocas iteraciones (menos de 10 normalmente), pero el número de veces que se debe realizar este cálculo es importante, especialmente cuando se trata de ajustar el caudal hasta conseguir que la temperatura de salida sea la deseada. Además, el trasiego de datos entre las estructuras que se han creado es importante dado que, como

hemos indicado previamente, se almacena bastante información sobre cada lazo. Consideramos que la optimización del código puede acometerse en un futuro, una vez que el código haya podido ser depurado en mayor profundidad.

Para otro tipo de simulaciones, como las que se explican en el apartado 4.1 relativo al análisis paramétrico, los tiempos de ejecución son despreciables al no trabajar con volúmenes de datos tan grandes.

3.4.3. Resultados de la verificación

Una vez que tenemos configurado SAM y nuestro programa de la misma manera ejecutamos SAM sobre un archivo con los datos meteorológicos de un año y en valores horarios, en concreto se trata de los datos del año 2007 que se emplearon para el estudio y anteproyecto de construcción de plantas reales emplazadas en el mismo lugar para el que hemos configurado la simulación.

En el campo solar de 120 lazos, con una superficie total de captación de $392400\ m^2$ la energía anual incidente es de 792,1 GWh, que se ve reducida a 686,4 GWh debido a que el ángulo de incidencia es distinto de cero (efecto coseno).

En la Tabla 3.5 se muestran los valores calculados por SAM y los que obtenemos con nuestro código. SAM denomina "Receiver thermal power incident" a la radiación solar que finalmente alcanza al absorbedor, es decir, la radaciación solar incidente menos las pérdidas ópticas. Este valor es equivalente a lo que nosotros hemos venido denominando potencia absorbida, \dot{q}_{abs}'' .

TABLA 3.5. Resultados globales anuales para las simulaciones con SAM y Python

Valores anuales	SAM	Código Python
Potencia incidente (GWh/año)	686	686
Potencia absorbida (GWh/año)	477	491
Rendimiento óptico	0,695	0,716
Pérdidas térmicas (GWh/año)	64	73
Rendimiento térmico	0,866	0,851
Potencia térmica (GWh/año)	413	418

Comprobamos que existe una ligera desviación en el rendimiento óptico anual (2,9 %) y en el rendimiento térmico anual (1,9 %). El origen de esta desviación se encuentra en la dificultad de adaptar exactamente la configuración de nuestra simulación a la de SAM pues no todos los parámetros son equivalentes en ambos casos. Por otro lado, por tratarse de valores anuales, existe una acumulación de pequeñas desviaciones que se producen entre ambas simulaciones, principalmente a primera y última hora del día y durante jornadas de gran inestabilidad en la radiación.

Nos fijaremos ahora en el comportamiento a lo largo de un día completo. Hemos seleccionado un día de gran estabilidad y buena radiación solar y otro día menos estable y con peores condiciones. En las figuras 3.19 a 3.21 podemos ver la evolución del caudal, la temperatura y la potencia térmica en ambas simulaciones para el día 17 de julio (buenas condiciones de radiación y estabilidad). En las figuras 3.22 a 3.24 podemos ver la evolución para el día 17 de junio, con peores condiciones tal y como se aprecia por los altibajos que presentan las curvas a lo largo del día. En todas las gráficas se ha representado DNI en el eje secundario.

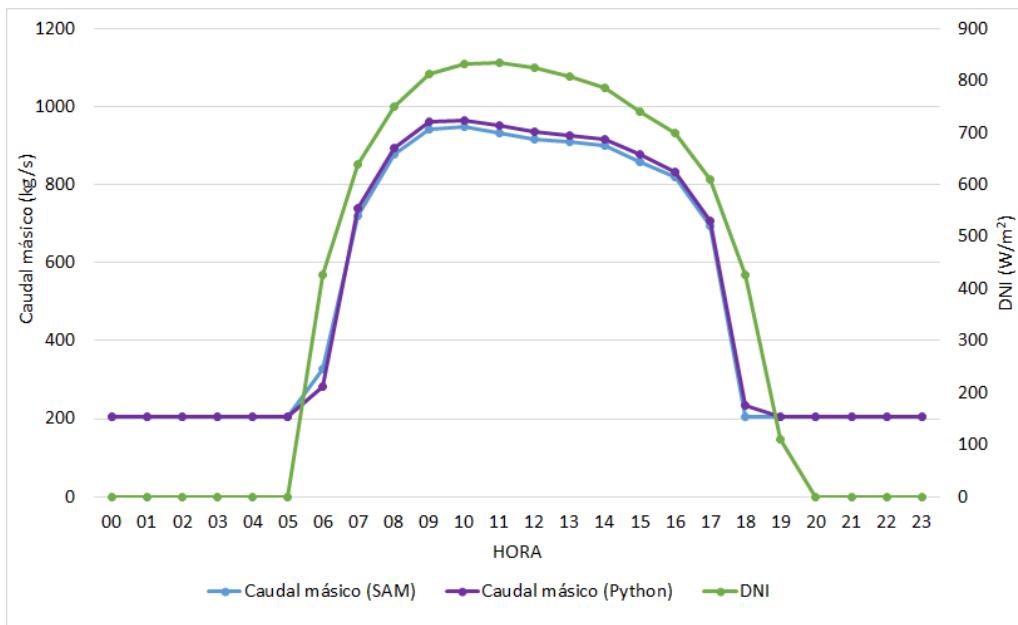


Fig. 3.19. Caudal másico calculado por SAM y por el código Python para el día 17/07/2007

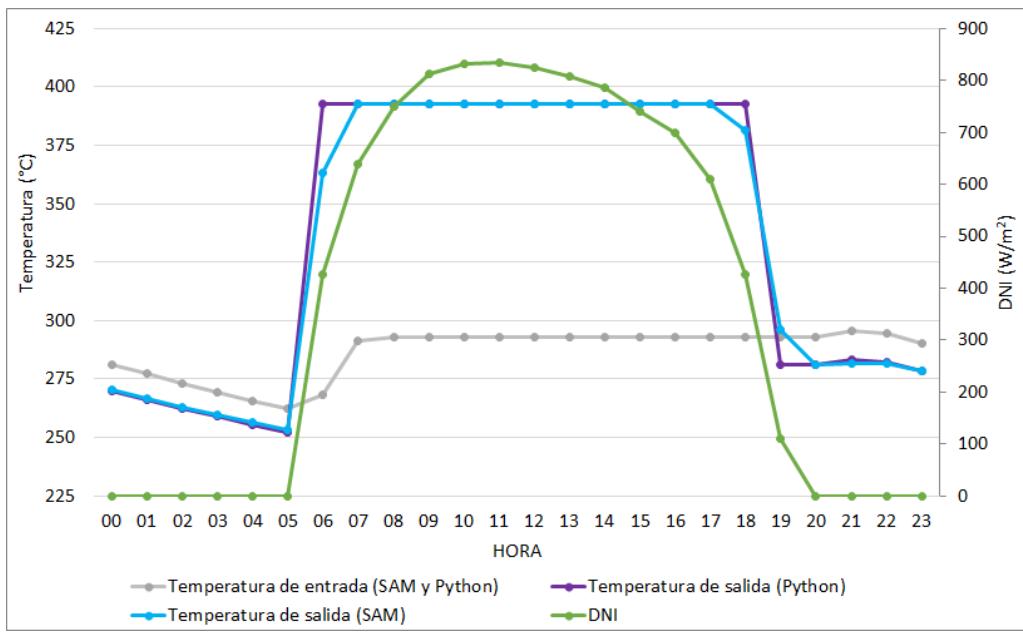


Fig. 3.20. Temperatura de entrada y de salida calculadas por SAM y por el código Python para el día 17/07/2007

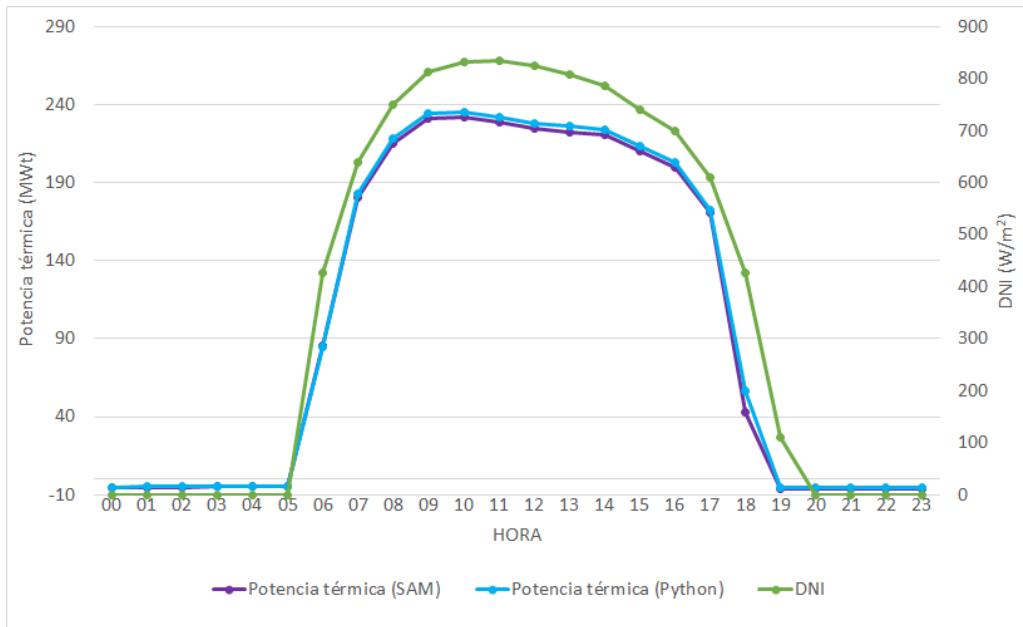


Fig. 3.21. Potencia térmica calculada por SAM y por el código Python para el día 17/07/2007

La tabla 3.6 muestra los datos de caudal, temperatura y potencia para los 24 registros horarios del día 17 de julio.

TABLA 3.6. Resultados de las simulaciones (S: SAM, P: Python) del día
17 de julio de 2007. Condiciones estables

Hora	DNI (S, P) (W/m ²)	Caudal (S) (kg/s)	Caudal (P) (kg/s)	T _{in} (S, P) (°C)	T _{out} (S) (°C)	T _{out} (P) (°C)	Pot. Térmica (S) (MWt)	Pot. Térmica (P) (MWt)
0:00	0	204,0	204,0	281,3	270,4	270,0	-5,5	-5,2
1:00	0	204,0	204,0	277,2	266,6	266,2	-5,3	-5,0
2:00	0	204,0	204,0	273,3	263,1	262,5	-5,1	-4,8
3:00	0	204,0	204,0	269,5	259,6	259,0	-4,9	-4,7
4:00	0	204,0	204,0	265,8	256,2	255,6	-4,7	-4,6
5:00	0	204,0	204,0	262,3	253,0	252,3	-4,6	-4,4
6:00	426	326,4	282,9	268,5	363,0	392,7	85,7	84,5
7:00	639	721,1	739,0	291,4	392,6	392,9	180,1	182,8
8:00	750	876,3	894,9	292,9	392,8	392,9	215,2	218,3
9:00	812	943,4	961,0	293,0	392,5	392,9	230,7	234,3
10:00	831	947,0	964,8	293,0	392,5	392,9	231,6	235,2
11:00	834	933,5	951,8	293,0	392,6	392,9	228,4	232,0
12:00	825	917,1	935,7	293,0	392,6	392,9	224,6	228,1
13:00	809	908,3	926,8	293,0	392,7	392,9	222,6	225,9
14:00	787	899,1	917,5	293,0	392,7	392,9	220,4	223,6
15:00	740	857,7	876,1	293,0	392,9	392,9	210,6	213,5
16:00	699	818,2	831,4	293,0	392,5	392,9	200,1	202,7
17:00	610	694,2	706,8	293,0	392,9	392,9	170,5	172,3
18:00	426	204,0	232,5	293,0	381,4	392,9	43,1	56,7
19:00	109	204,0	204,0	293,0	295,9	281,0	-6,4	-5,6
20:00	0	204,0	204,0	293,0	280,9	281,0	-6,2	-5,6
21:00	0	204,0	204,0	295,7	281,4	283,5	-6,3	-5,7
22:00	0	204,0	204,0	294,6	281,9	282,5	-6,3	-5,6
23:00	0	204,0	204,0	290,5	278,6	278,6	-6,0	-5,5

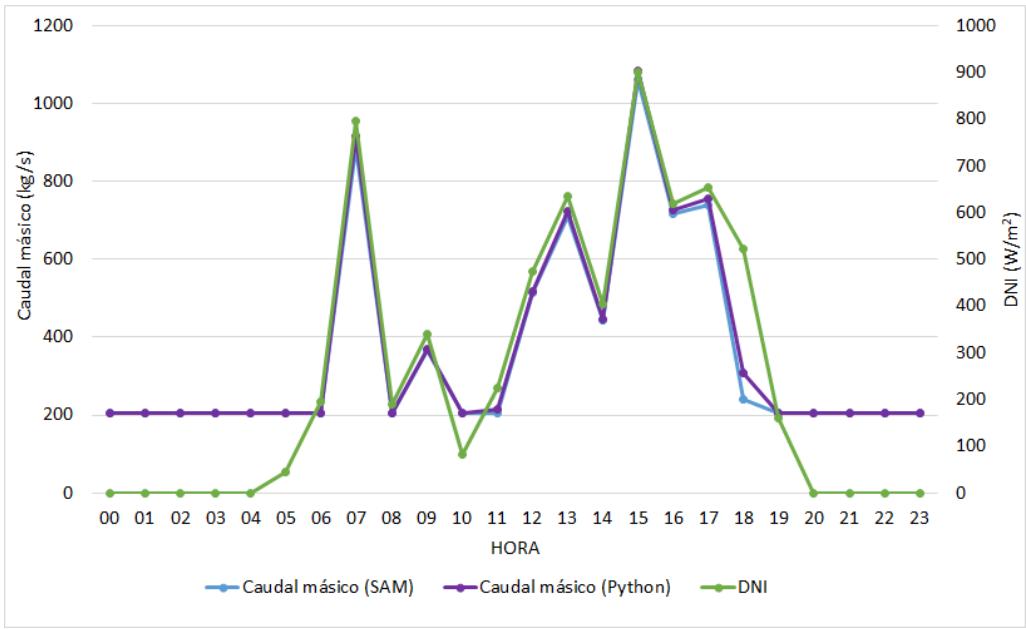


Fig. 3.22. Caudal másico calculado por SAM y por el código Python para el día 17/06/2007

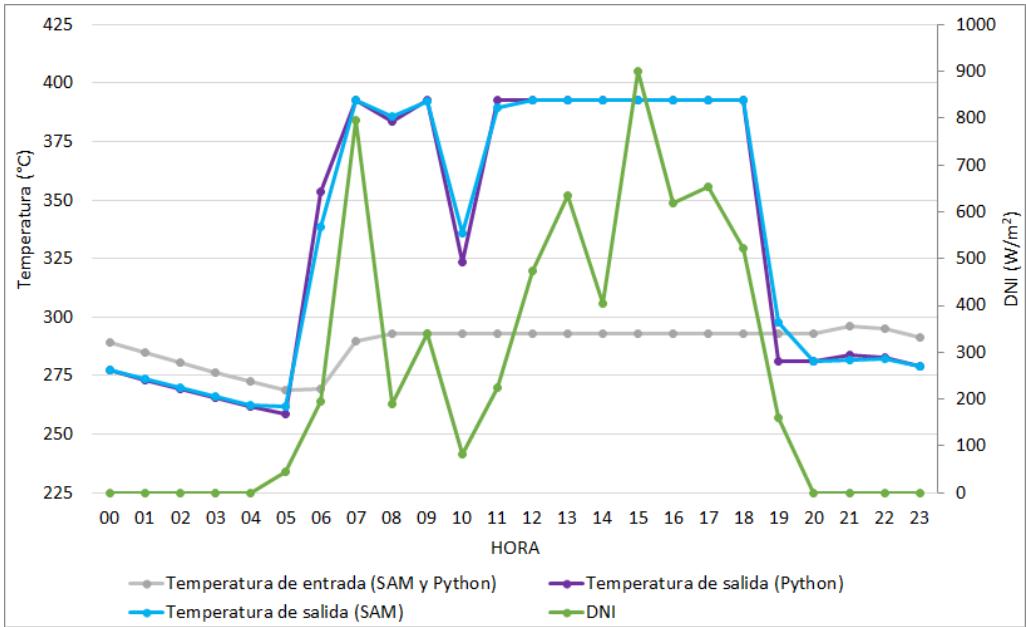


Fig. 3.23. Temperatura de entrada y de salida calculadas por SAM y por el código Python para el día 17/06/2007

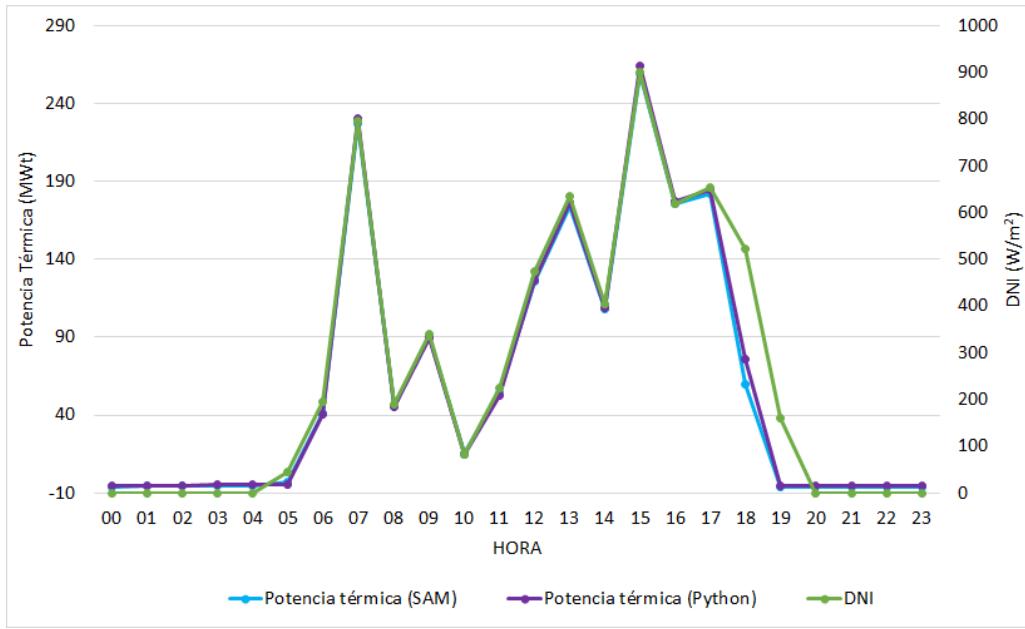


Fig. 3.24. Potencia térmica calculada por SAM y por el código Python para el día 17/06/2007

En la tabla 3.7 se muestran los datos correspondientes al día 17 de junio de 2007.

TABLA 3.7. Resultados de las simulaciones (S: SAM, P: Python) para el día 17 de junio de 2007. Condiciones inestables

Hora	DNI (S, P) (W/m ²)	Caudal (S) (kg/s)	Caudal (P) (kg/s)	T_{in} (S, P) (°C)	T_{out} (S) (°C)	T_{out} (P) (°C)	Pot. Térmica (S) (MWt)	Pot. Térmica (P) (MWt)
0:00	0	204,0	204,0	289,3	277,5	277,4	-6,0	-5,5
1:00	0	204,0	204,0	284,8	273,5	273,3	-5,8	-5,3
2:00	0	204,0	204,0	280,6	269,6	269,3	-5,5	-5,1
3:00	0	204,0	204,0	276,5	265,9	265,5	-5,3	-5,0
4:00	0	204,0	204,0	272,5	262,3	261,8	-5,1	-4,8
5:00	45	204,0	204,0	268,8	262,0	258,4	-3,2	-4,7
6:00	194	204,0	204,0	269,2	338,7	353,3	41,5	40,3
7:00	795	885,8	916,9	289,7	392,7	392,9	227,1	230,5
8:00	189	204,0	204,0	292,8	385,9	383,5	46,1	44,9
9:00	339	366,8	367,7	292,9	392,2	392,9	90,0	89,7
10:00	81	204,0	204,0	293,0	336,0	323,7	15,6	14,7

Tabla 3.7 continúa desde la página anterior

Hora	DNI (S, P) (W/m ²)	Caudal (S) (kg/s)	Caudal (P) (kg/s)	T_{in} (S, P) (°C)	T_{out} (S) (°C)	T_{out} (P) (°C)	Pot. Térmica (S) (MWt)	Pot. Térmica (P) (MWt)
11:00	225	204,0	215,4	293,0	389,6	392,9	53,8	52,5
12:00	473	514,8	518,7	293,0	392,6	392,9	125,8	126,4
13:00	634	711,7	722,5	293,0	392,5	392,9	174,1	176,1
14:00	403	442,9	445,2	293,0	392,5	392,9	108,4	108,5
15:00	899	1059,6	1082,5	293,0	392,7	392,9	259,6	263,9
16:00	619	716,1	727,6	293,0	392,7	392,9	175,4	177,4
17:00	654	740,2	755,1	293,0	393,0	392,9	181,9	184,1
18:00	522	241,7	309,5	293,0	392,7	392,9	59,4	75,4
19:00	160	204,0	204,0	293,0	297,8	280,9	-6,5	-5,6
20:00	0	204,0	204,0	293,0	281,0	280,9	-6,2	-5,6
21:00	0	204,0	204,0	296,2	281,6	283,8	-6,4	-5,7
22:00	0	204,0	204,0	295,3	282,4	282,9	-6,4	-5,7
23:00	0	204,0	204,0	291,1	279,1	279,1	-6,1	-5,6

En vista a los resultados de la comparación con SAM, consideramos que nuestro código obtiene valores adecuados y cuenta con la flexibilidad suficiente que permite emplearlo en la simulación del comportamiento de concentradores cilindroparabólicos. En el siguiente capítulo realizaremos algunos análisis a modo de ejemplo.

4. APLICACIONES DEL CÓDIGO DE SIMULACIÓN

4.1. Aplicación para el análisis paramétrico

Una vez que hemos confirmado que nuestro código es una herramienta válida para la simulación de un campo solar real, nos proponemos, a continuación, aprovecharlo para analizar el comportamiento de los componentes del campo solar bajo diferentes condiciones o con diferentes configuraciones. Este tipo de análisis es de especial utilidad durante la fase de diseño, cuando deben seleccionarse los componentes del sistema con el fin de alcanzar unos objetivos de rendimiento o potencia generada.

4.1.1. Rendimiento del HCE en función de la radiación normal directa, *DNI*

En la tabla 4.1 se muestran los coeficientes para el cálculo de la emisividad ε_{ext} conforme a la ec.(3.12) y el valor de la absorbtividad de algunos modelos de HCE según se recogen en [1], incluido el modelo NREL#6, que en realidad es un modelo de HCE con un recubrimiento selectivo teórico propuesto para desarrollos futuros.

TABLA 4.1. Constantes del modelo de emisividad equivalente para cada uno de los receptores seleccionados.

Receptor	A_0	A_1	Absortividad (%)
Solel UVAC 2/2008	1,31E-04	1,01E-01	97
Solel UVAC 3/2010	2,06E-04	4,30E-02	96
Schott PTR70	1,82E-04	8,61E-02	95
Schott PTR70/2008	1,43E-04	3,45E-02	95,5
SkyFuel SkyTrough DSP	1,48E-04	4,00E-02	95 ¹
ASE HEMS08	2,03E-04	-1,03E-02	95
NREL #6	1,52E-04	1,96E-03	96

Si simulamos el comportamiento de un único HCE de cada modelo (considerando

¹Al no disponerse de este dato, se ha empleado este valor con el fin de poder incluir el modelo en el test.

Fuente [1]

que todos tienen la misma longitud de 4,05 m), para diferentes valores de DNI y bajo las mismas condiciones de caudal y temperatura de entrada del HTF, podemos realizar una comparativa de los rendimientos. Salvo que se diga lo contrario, en adelante emplearemos los mismos datos geográficos ya indicados cuando realizamos la simulación de verificación con SAM. Para la fecha y hora de simulación se empleará el 1 de julio a las 12:00 UTC (14:00 hora local).

A modo de ejemplo, se muestra el código 4.1 desarrollado para realizar este tipo de simulación a partir de un archivo de configuración *test_1.json*. Este archivo es similar al mostrado en el apartado anterior, con las siguientes salvedades:

- Solo es necesario indicar la configuración de un solo lazo con un único SCA que contiene un único HCE. De este modo, simulamos solo un HCE de 4,05 m.
- El archivo de origen de datos contiene varios registros con los mismos datos de fecha, hora, temperatura ambiente y viento, pero varía la DNI desde 100 hasta 1000 W/m^2 .

El programa se dispone a hacer una simulación con la configuración indicada para cada registro del archivo de origen de datos, pero también realiza un bucle recorriendo todos los modelos de HCE que existen en el archivo *HCE_library.json*. De esta forma, al finalizar, tenemos una tabla con el rendimiento de cada modelo para cada uno de los valores de DNI. Se hace uso de la Clase *LoopSimulation* creada expresamente con el fin de facilitar simulaciones con un solo lazo de configuración variable.

CÓDIGO 4.1. Programa para el análisis del rendimiento en función de DNI

```
1 import csenergy as cs
2 import pandas as pd
3 import json
4 from datetime import datetime
5
6 FLAG_00 = datetime.now()
7 with open("./saved_configurations/test_1.json") as simulation_file:
8     simulation_settings = json.load(simulation_file)
9 with open("./hce_files/HCE_library.json") as hces_file:
10    hces_configurations = json.load(hces_file)
11 dict_resultados = {}
```

```

12 for hce_conf in hces_configurations:
13     simulation_settings['HCE'].update(hce_conf)
14     dni_index = []
15     simulation = cs.LoopSimulation(simulation_settings)
16     dict_resultados[hce_conf['Name']] = []
17     for row in simulation.datasource.dataframe.iterrows():
18         solarpos = simulation.site.get_solarposition(row)
19         aoi = simulation.base_loop.scas[0].get_aoi(solarpos)
20         values = {'tin': 573,
21                   'pin': 1900000,
22                   'massflow': 6}
23         simulation.base_loop.initialize('values', values)
24         simulation.base_loop.calc_loop_pr_for_massflow(
25             row,
26             solarpos,
27             simulation.htf,
28             simulation.model)
29         tout = simulation.base_loop.scas[-1].hces[-1].tout
30         pout = simulation.base_loop.scas[-1].hces[-1].pout
31         simulation.base_loop.set_loop_values_from_HCEs()
32         pr = simulation.base_loop.pr
33         dict_resultados[hce_conf['Name']].append(pr)
34         dni_index.append(row[1]['DNI'])
35         simulation.datasource.dataframe.at[row[0], 'pr'] = pr
36         simulation.datasource.dataframe.at[row[0], 'tout'] = tout
37         simulation.datasource.dataframe.at[row[0], 'pout'] = pout
38     dfsalida = pd.DataFrame(dict_resultados, index=dni_index)
39     dfsalida.to_csv('rendimiento_dni.csv', sep=';', decimal=',')
40     FLAG_01 = datetime.now()
41     DELTA_01 = FLAG_01 - FLAG_00
42     print("Total runtime: ", DELTA_01.total_seconds())

```

El tiempo total de ejecución es de 1,375 segundos. El archivo *rendimiento_dni.csv* contiene los datos volcados a la finalización del programa.

En la tabla 4.2 se muestran los resultados de los rendimientos calculados para cada modelo de HCE con una temperatura de entrada de 300 °C y un caudal de 6 kg/s.

TABLA 4.2. Rendimiento en función de la radiación normal incidente
para distintos modelos de HCE

DNI (W/m ²)	Schott PTR70	Schott PTR70 2008	Solel UVAC 2	Solel UVAC 3	SkyFuel SkyTrough DSP	ASE HEMS08	NREL #6
100	0,521	0,728	0,521	0,638	0,837	0,818	0,834
200	0,757	0,862	0,757	0,816	0,917	0,907	0,916
300	0,837	0,907	0,837	0,876	0,944	0,938	0,943
400	0,876	0,930	0,877	0,907	0,958	0,953	0,957
500	0,900	0,944	0,900	0,925	0,966	0,962	0,965
600	0,917	0,953	0,917	0,937	0,971	0,968	0,971
700	0,928	0,959	0,928	0,945	0,975	0,972	0,975
800	0,937	0,964	0,937	0,952	0,978	0,976	0,978
900	0,943	0,968	0,943	0,957	0,980	0,978	0,980
1000	0,948	0,971	0,949	0,961	0,982	0,980	0,982

En la Fig.4.1 se muestra gráficamente cómo evoluciona el rendimiento con el aumento de *DNI*. Se aprecia cómo los modelos de última generación presentan un buen comportamiento incluso con bajas radiaciones.

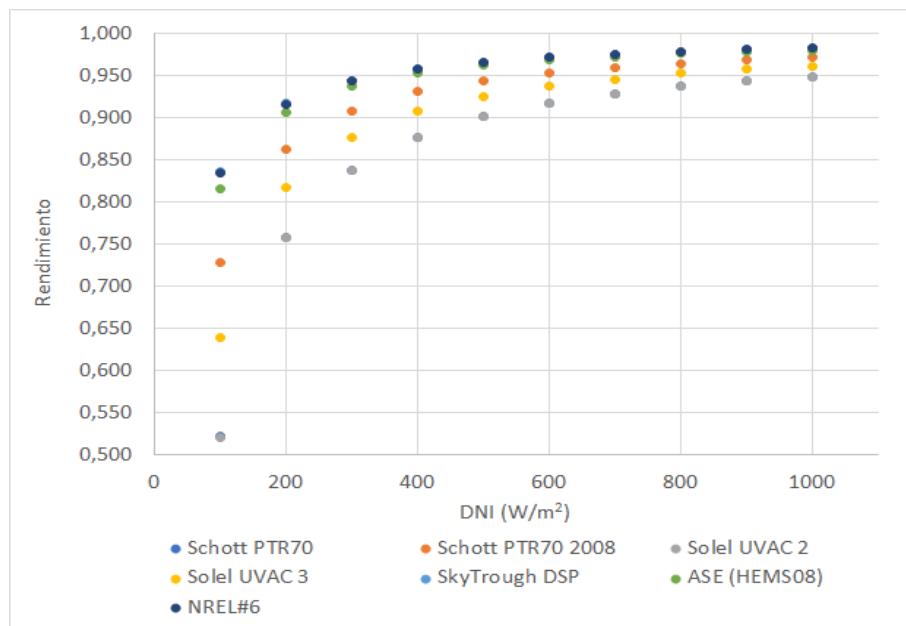


Fig. 4.1. Rendimiento térmico en función de DNI para diferentes modelos de HCE.

$$T_{in}=300 \text{ }^{\circ}\text{C}, \dot{m} = 6 \text{ kg/s}$$

4.1.2. Rendimiento del HCE en función de la temperatura de entrada

Otra decisión importante que debe tomarse a la hora del diseño de un campo solar es la temperatura nominal de operación. Intervienen varios criterios, como la temperatura y el caudal de fluido caloportador que demanda el bloque de potencia así como las limitaciones que impone el propio fluido a fin de evitar su degradación. Un análisis del rendimiento del HCE en función de la temperatura de operación puede ayudar en la toma de decisiones y en la selección del modelo más adecuado.

El procedimiento es muy parecido al caso anterior tal y como se muestra en el 4.2, que mostramos a continuación a modo de ejemplo², solo que en esta ocasión lo que se hace es ir modificando la temperatura de entrada.

CÓDIGO 4.2. Programa para el análisis del rendimiento en función de T_{in}

```
1 with open("./saved_configurations/test_2.json") as simulation_file:
2     simulation_settings = json.load(simulation_file)
3 with open("./hce_files/HCE_library.json") as hces_file:
4     hces_configurations = json.load(hces_file)
5 dict_resultados = {}
6 for hce_conf in hces_configurations:
7     simulation_settings['HCE'].update(hce_conf)
8     tin_index = []
9     simulation = cs.LoopSimulation(simulation_settings)
10    dict_resultados[hce_conf['Name']] = []
11    for row in simulation.datasource.datasource.iterrows():
12        solarpos = simulation.site.get_solarposition(row)
13        aoi = simulation.base_loop.scas[0].get_aoi(solarpos)
14        for tin in range(473, 693, 10):
15            values = {'tin': tin,
16                      'pin': 1900000,
17                      'massflow': 6}
18            simulation.base_loop.initialize('values', values)
19            simulation.base_loop.calc_loop_pr_for_massflow(
20                row,
21                solarpos,
```

²El resto de *scripts* para los diferentes test realizados en este capítulo pueden encontrarse en el archivo que contiene la versión electrónica de este TFG

```

22         simulation.htf,
23         simulation.model)
24     tout = simulation.base_loop.scas[-1].hces[-1].tout
25     pout = simulation.base_loop.scas[-1].hces[-1].pout
26     simulation.base_loop.set_loop_values_from_HCEs()
27     pr = simulation.base_loop.pr
28     dict_resultados[hce_conf['Name']].append(pr)
29     tin_index.append(tin-273)
30     simulation.datasource.dataframe.at[row[0], 'pr'] = pr
31     simulation.datasource.dataframe.at[row[0], 'tout'] =
32         tout
33     simulation.datasource.dataframe.at[row[0], 'pout'] =
34         pout
35     dfsalida = pd.DataFrame(dict_resultados, index=tin_index)
36     print(dfsalida)
37     dfsalida.to_csv('rendimiento_temperatura.csv', sep=';', decimal=',')

```

En la tabla 4.3 se muestran los resultado de simular el funcionamiento de varios modelos de HCE con un caudal constante de fluido caloportador a diferentes temperaturas de entrada.

TABLA 4.3. Rendimiento en función de la temperatura de entrada del HTF para diferentes modelos de HCE

$T_{in}(^{\circ}C)$	Schott PTR70	Schott PTR70 2008	Solel UVAC 2	Solel UVAC 3	SkyFuel SkyTrough DSP	ASE HEMS08	NREL #6
200	0,975	0,986	0,974	0,982	0,993	0,993	0,993
210	0,972	0,985	0,971	0,980	0,992	0,992	0,992
220	0,969	0,983	0,969	0,978	0,991	0,990	0,991
230	0,966	0,981	0,965	0,976	0,990	0,989	0,990
240	0,963	0,979	0,962	0,973	0,988	0,988	0,988
250	0,959	0,977	0,958	0,970	0,987	0,986	0,987
260	0,955	0,975	0,955	0,967	0,986	0,984	0,985
270	0,951	0,973	0,951	0,964	0,984	0,982	0,984
280	0,946	0,970	0,946	0,960	0,982	0,980	0,982
290	0,942	0,967	0,942	0,956	0,980	0,978	0,980

Tabla 4.3 continúa desde la página anterior

$T_{in}(^{\circ}C)$	Schott PTR70	Schott PTR70 2008	Solel UVAC 2	Solel UVAC 3	SkyFuel SkyTrough DSP	ASE HEMS08	NREL #6
300	0,937	0,964	0,937	0,952	0,978	0,976	0,978
310	0,931	0,961	0,931	0,947	0,976	0,973	0,975
320	0,925	0,957	0,926	0,943	0,974	0,970	0,973
330	0,919	0,953	0,920	0,938	0,971	0,967	0,970
340	0,912	0,949	0,914	0,932	0,968	0,963	0,967
350	0,905	0,945	0,907	0,926	0,965	0,960	0,964
360	0,898	0,941	0,900	0,920	0,962	0,956	0,961
370	0,890	0,936	0,892	0,913	0,958	0,951	0,957
380	0,881	0,931	0,884	0,906	0,955	0,947	0,954
390	0,872	0,925	0,876	0,898	0,951	0,942	0,949
400	0,863	0,919	0,867	0,890	0,946	0,937	0,945
410	0,853	0,913	0,858	0,882	0,942	0,931	0,941

La representación gráfica evidencia la caída de rendimiento según aumenta la temperatura de entrada aunque, nuevamente, vemos que los modelos más modernos mantienen un mejor rendimiento a temperaturas más elevadas.

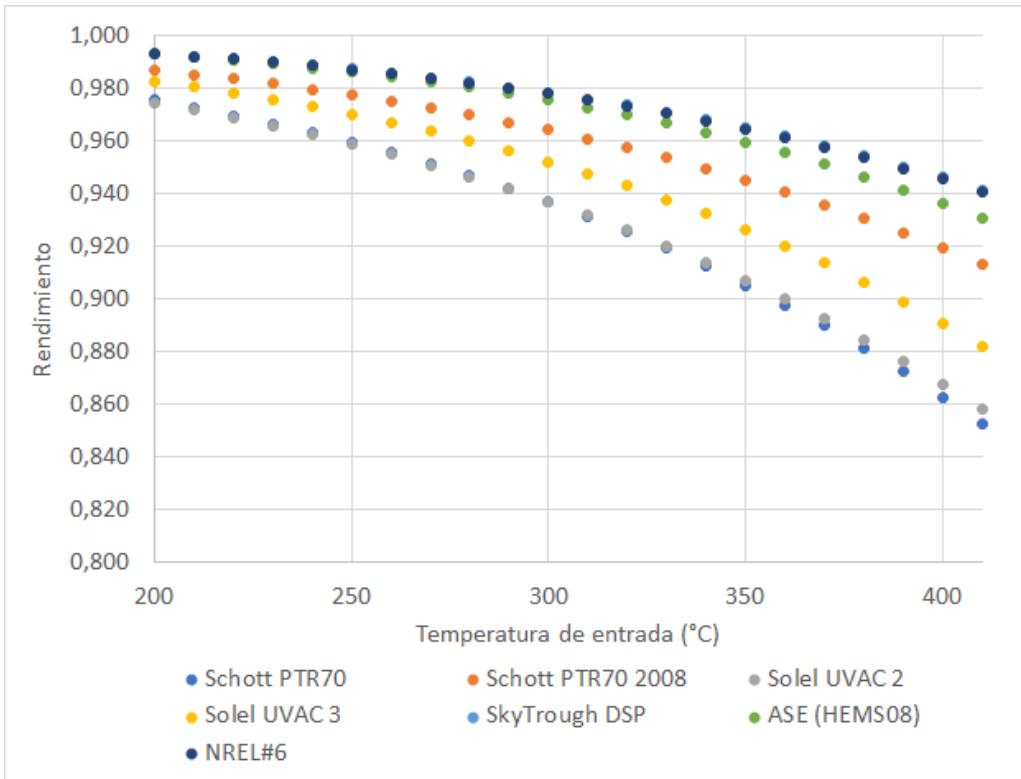


Fig. 4.2. Rendimiento térmico en función de la temperatura de entrada del HTF para diferentes modelos de HCE. $DNI = 800W/m^2$, $\dot{m} = 6 \text{ kg/s}$

4.1.3. Rendimiento del HCE en función del flujo de radiación absorbido, \dot{q}_{abs}''

El flujo de radiación absorbido, \dot{q}_{abs}'' , también es un factor determinante en el diseño del concentrador solar. Es interesante comprobar que el rendimiento aumenta según lo hace \dot{q}_{abs}'' pero, tal y como se aprecia en la Fig. 4.3, el Modelo de 4º Orden recoge la presencia de un máximo que no es recogido por los modelos de menos precisos. En todo caso, este máximo se encuentra para valores de flujo máximo muy superiores a los que se alcanzan con factores de concentración propios de CCP.

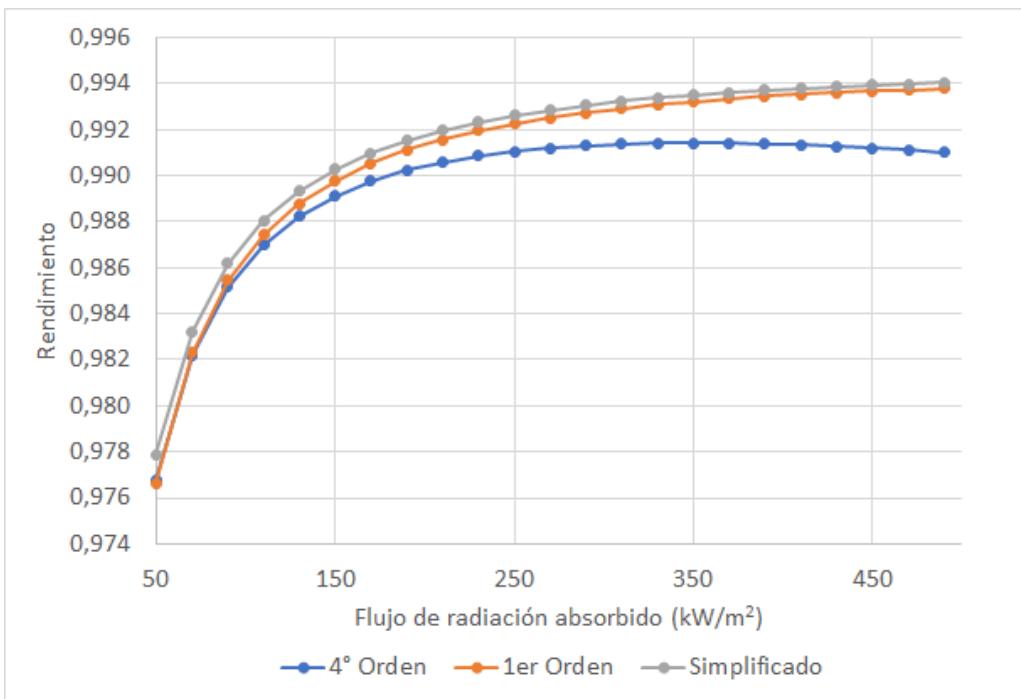


Fig. 4.3. Rendimiento térmico en función del flujo de radiación absorbido para diferentes modelos de HCE. $t_{in} = 300 \text{ }^{\circ}\text{C}$, $\dot{m} = 6 \text{ kg/s}$

Los valores correspondientes se muestran en la tabla 4.4.

TABLA 4.4. Rendimiento en función del flujo de radiación absorbido y para cada modelo teórico

\dot{q}_{abs}''	4º Orden	1 ^{er} Orden	Simplificado
50000	0,9768	0,9766	0,9779
70000	0,9822	0,9823	0,9832
90000	0,9852	0,9854	0,9862
110000	0,9870	0,9874	0,9881
130000	0,9882	0,9888	0,9893
150000	0,9891	0,9898	0,9903
170000	0,9898	0,9905	0,9910
190000	0,9902	0,9911	0,9915
210000	0,9906	0,9916	0,9920
230000	0,9908	0,9919	0,9923
250000	0,9910	0,9923	0,9926
270000	0,9912	0,9925	0,9929

Tabla 4.4 continúa desde la página anterior

\dot{q}_{abs}''	4° Orden	1 ^{er} Orden	Simplificado
290000	0,9913	0,9927	0,9931
310000	0,9914	0,9929	0,9932
330000	0,9914	0,9931	0,9934
350000	0,9914	0,9932	0,9935
370000	0,9914	0,9933	0,9936
390000	0,9914	0,9934	0,9937
410000	0,9913	0,9935	0,9938
430000	0,9913	0,9936	0,9939
450000	0,9912	0,9937	0,9939
470000	0,9911	0,9937	0,9940
490000	0,9910	0,9938	0,9940

4.1.4. Simulación con los diferentes modelos teóricos

Comparamos ahora el resultado de simular con los tres modelos: modelo de 4º Orden, modelo de 1^{er} Orden y modelo Simplificado. En las siguientes figuras podemos comparar los resultados de temperatura, caudal, rendimiento y potencia térmica de la configuración de campo solar empleada en el apartado 3.4.1 para un día del año (se ha tomado el día 2 de marzo por tener buenas condiciones de radiación y estabilidad).

TABLA 4.5. Temperaturas obtenidas en la simulación con cada modelo teórico. Datos del día 2/3/2007. Condiciones estables y buena radiación

Hora	DNI (W/m ²)	T_{in} (°C)	T_{out} (SAM) (°C)	T_{out} (4ºOrd.) (°C)	T_{out} (1 ^{er} Ord.) (°C)	T_{out} (Simplif.) (°C)
0:00	0	277	266	266	266	261
1:00	0	273	262	262	262	258
2:00	0	269	259	258	258	254
3:00	0	265	255	255	255	251
4:00	0	262	252	251	251	248
5:00	0	258	249	248	248	245

Tabla 4.5 continúa desde la página anterior

Hora	DNI (W/m ²)	T_{in} (°C)	T_{out} (SAM) (°C)	T_{out} (4 ^o Ord.) (°C)	T_{out} (1 ^{er} Ord.) (°C)	T_{out} (Simplif.) (°C)
6:00	0	255	246	245	245	242
7:00	234	252	263	243	243	240
8:00	596	272	377	393	393	393
9:00	724	292	392	393	393	393
10:00	806	293	393	393	393	393
11:00	859	293	393	393	393	393
12:00	876	293	393	393	393	393
13:00	867	293	393	393	393	393
14:00	846	293	393	393	393	393
15:00	781	293	393	393	393	393
16:00	663	293	393	393	393	393
17:00	420	293	313	281	281	275
18:00	1	293	284	281	281	275
19:00	0	298	282	285	285	280
20:00	0	298	284	285	285	279
21:00	0	293	281	281	281	276
22:00	0	289	277	277	277	272
23:00	0	284	273	273	273	268

El resultado gráfico de este test puede verse en la figura Fig.4.4, donde se aprecia que todos los modelos encuentran soluciones similares en la simulación. El resultado era de esperar pues las temperaturas de trabajo, inferiores a 400 °C, están dentro del rango de validez de aplicación de todos los modelos. Otra vez los resultados son muy parecidos a los obtenidos con SAM.

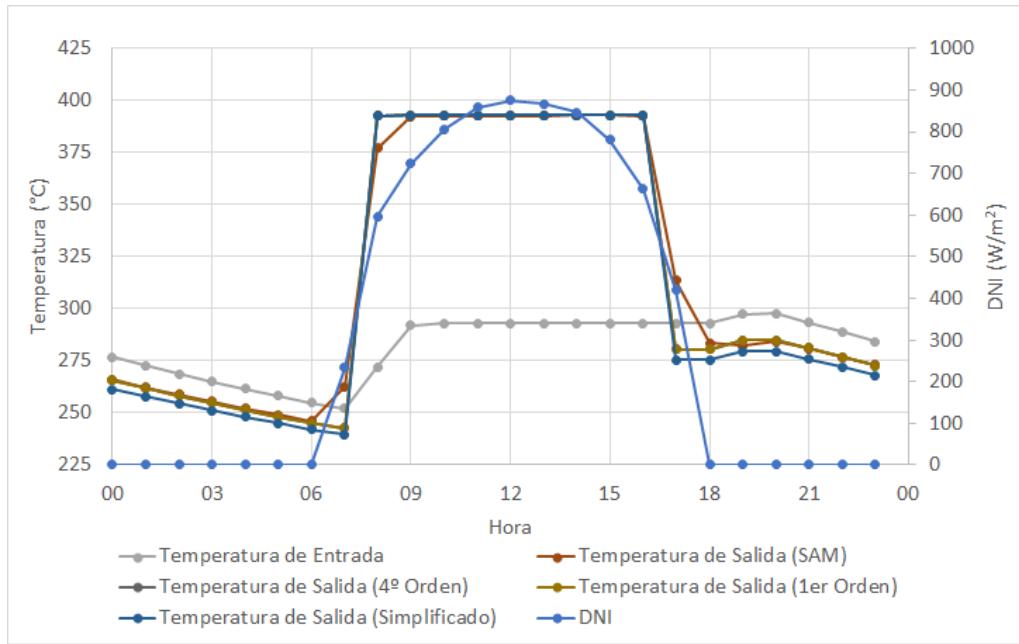


Fig. 4.4. Temperaturas de salida obtenidas con los tres modelos. Simulación con los datos del día 2/3/2007.

En el caso de los caudales y la potencia térmica los resultados son similares, tal y como se aprecia en las figuras 4.5 y 4.6 respectivamente. Los valores numéricos se muestran en las tablas 4.6 y 4.7.

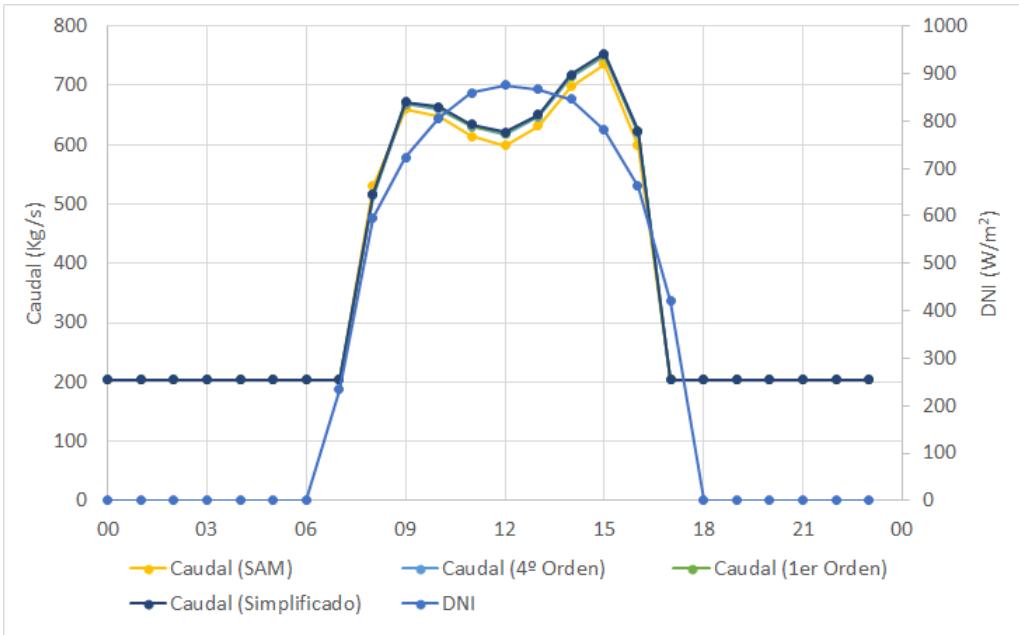


Fig. 4.5. Caudales de salida obtenidos con los tres modelos. Datos del día 2/3/2007.

Condiciones estables y buena radiación

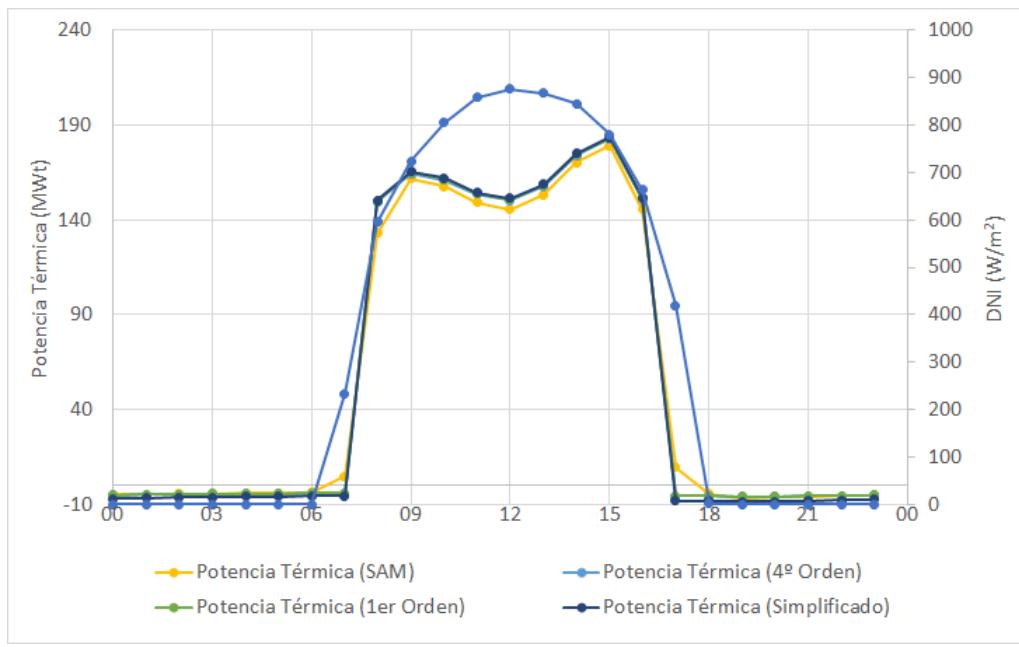


Fig. 4.6. Potencia térmica obtenida con cada uno de los tres modelos. Datos del día 2/3/2007. Condiciones estables y buena radiación

TABLA 4.6. Caudales obtenidos en la simulación para cada modelo. Datos del día 2/3/2007. Condiciones estables y buena radiación

Hora	DNI (W/m^2)	Caudal	Caudal	Caudal	Caudal
		(SAM)	(4º Ord.)	(1º Ord.)	(Simplif.)
0:00	0	204	204	204	204
1:00	0	204	204	204	204
2:00	0	204	204	204	204
3:00	0	204	204	204	204
4:00	0	204	204	204	204
5:00	0	204	204	204	204
6:00	0	204	204	204	204
7:00	234	204	204	204	204
8:00	596	530	514	516	518
9:00	724	660	669	670	672
10:00	806	648	661	663	664
11:00	859	614	630	632	634
12:00	876	599	617	619	621

Tabla 4.6 continúa desde la página anterior

Hora	DNI (W/m ²)	Caudal (SAM) (kg/s)	Caudal (4 ^o Ord.) (kg/s)	Caudal (1 ^{er} Ord.) (kg/s)	Caudal (Simplif.) (kg/s)
13:00	867	632	648	650	651
14:00	846	699	714	716	718
15:00	781	736	750	752	753
16:00	663	599	620	622	624
17:00	420	204	204	204	204
18:00	1	204	204	204	204
19:00	0	204	204	204	204
20:00	0	204	204	204	204
21:00	0	204	204	204	204
22:00	0	204	204	204	204
23:00	0	204	204	204	204

TABLA 4.7. Potencia térmica calculada en la simulación de cada modelo.

Datos del día 2/3/2007. Condiciones estables y buena radiación

Hora	DNI (W/m ²)	P _{th} (SAM) (MWt)	P _{th} (4 ^o Ord.) (MWt)	P _{th} (1 ^{er} Ord.) (MWt)	P _{th} (Simplif.) (MWt)
0:00	0	-4,9	-5,0	-5,0	-6,9
1:00	0	-4,7	-4,9	-4,9	-6,7
2:00	0	-4,5	-4,8	-4,8	-6,5
3:00	0	-4,4	-4,6	-4,6	-6,3
4:00	0	-4,2	-4,5	-4,5	-6,0
5:00	0	-4,1	-4,4	-4,4	-5,9
6:00	0	-3,9	-4,3	-4,3	-5,7
7:00	234	4,7	-4,2	-4,2	-5,6
8:00	596	133,1	149,5	150,1	150,5
9:00	724	161,6	164,6	165,0	165,4
10:00	806	157,7	161,2	161,6	162,1

Tabla 4.7 continúa desde la página anterior

Hora	DNI (W/m ²)	P_{th}	P_{th}	P_{th}	P_{th}
		(SAM) (MWt)	(4 ^o Ord.) (MWt)	(1 ^{er} Ord.) (MWt)	(Simplif.) (MWt)
11:00	859	149,2	153,6	154,1	154,5
12:00	876	145,5	150,5	150,9	151,3
13:00	867	153,4	157,9	158,3	158,8
14:00	846	170,2	174,2	174,6	175,0
15:00	781	179,3	182,7	183,2	183,6
16:00	663	145,6	151,1	151,6	152,0
17:00	420	9,7	-5,7	-5,7	-8,2
18:00	1	-4,4	-5,7	-5,7	-8,1
19:00	0	-7,1	-5,9	-5,9	-8,4
20:00	0	-6,3	-5,9	-5,9	-8,4
21:00	0	-5,8	-5,7	-5,7	-8,1
22:00	0	-5,5	-5,5	-5,5	-7,8
23:00	0	-5,3	-5,4	-5,4	-7,5

4.1.5. Simulación cambiando el tamaño de la malla de integración

En el caso de tener que realizar un gran número de simulaciones, el tiempo de cálculo puede ser un factor limitante. Una forma de reducirlo es aumentando el tamaño de la malla de integración. Esto es equivalente a considerar un HCE de tamaño mayor al real, con lo que el número de cálculos por lazo se reduce. Realizamos una simulación con cada modelo teórico en la que calculamos el rendimiento térmico de un lazo completo variando el tamaño de malla de integración.

En las figuras 4.7 a 4.10 se aprecia una mayor divergencia en el rendimiento calculado por cada modelo a medida que aumenta el tamaño de malla. Las diferencias son mayores para valores más altos de DNI. Como conclusión, podemos aceptar tamaños de malla de no más de 100 m para el Modelo de 4º Orden. Para el Modelo de 1^{er} Orden y el Modelo Simplificado vemos que una malla de unos 50 m ya empieza a mostrar diferencias apreciables respecto al modelo de 4º Orden.

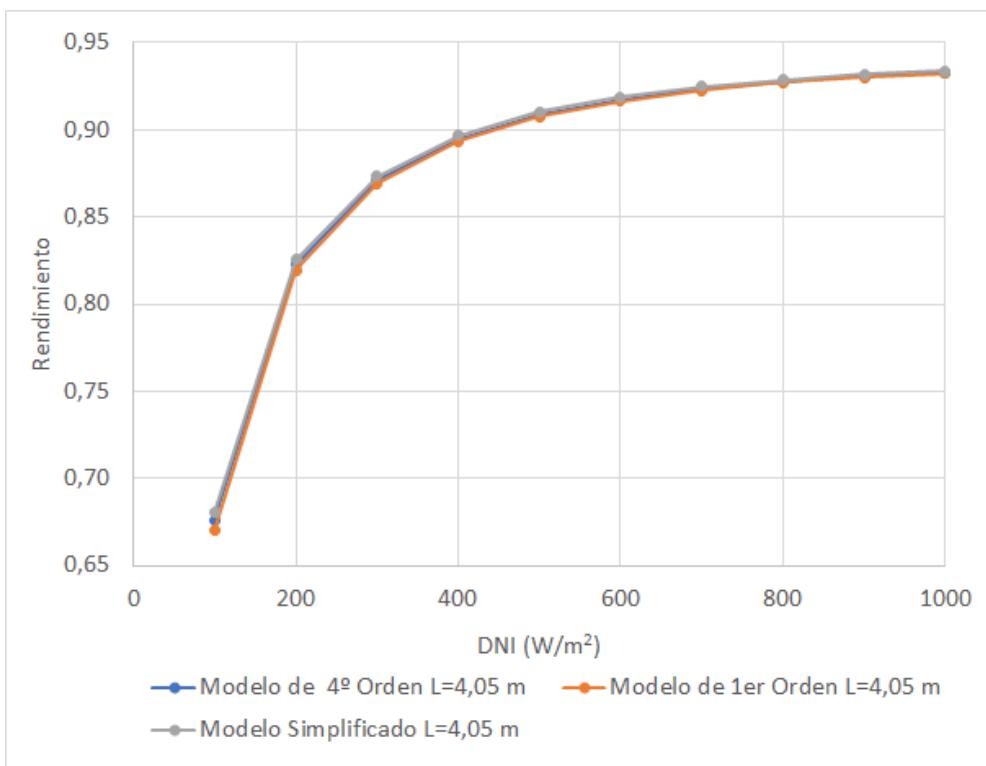


Fig. 4.7. Rendimiento calculado con cada modelo para un tamaño de malla de integración de 4,05 m

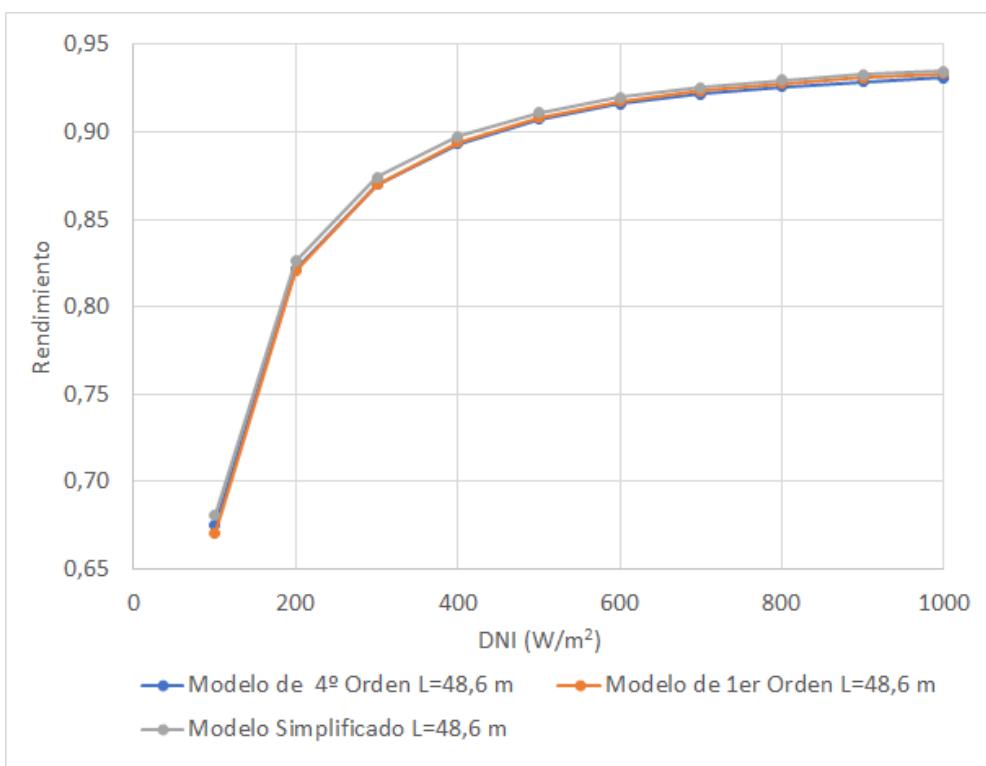


Fig. 4.8. Rendimiento calculado con cada modelo para un tamaño de malla de integración de 48,60 m

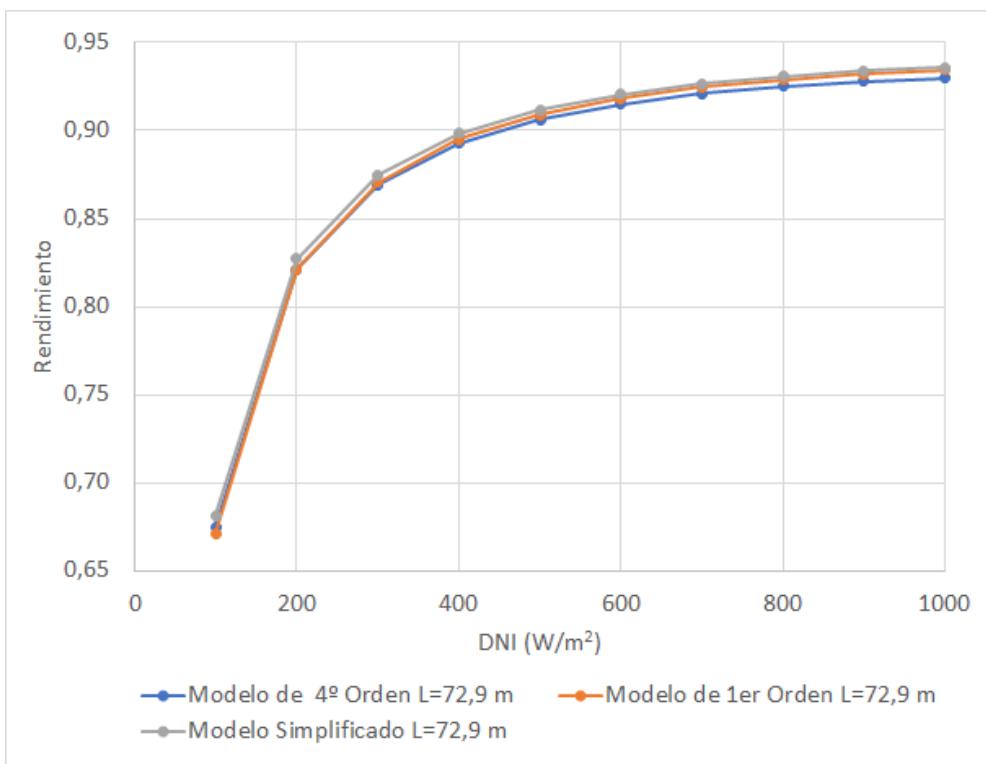


Fig. 4.9. Rendimiento calculado con cada modelo para un tamaño de malla de integración de 72,90 m

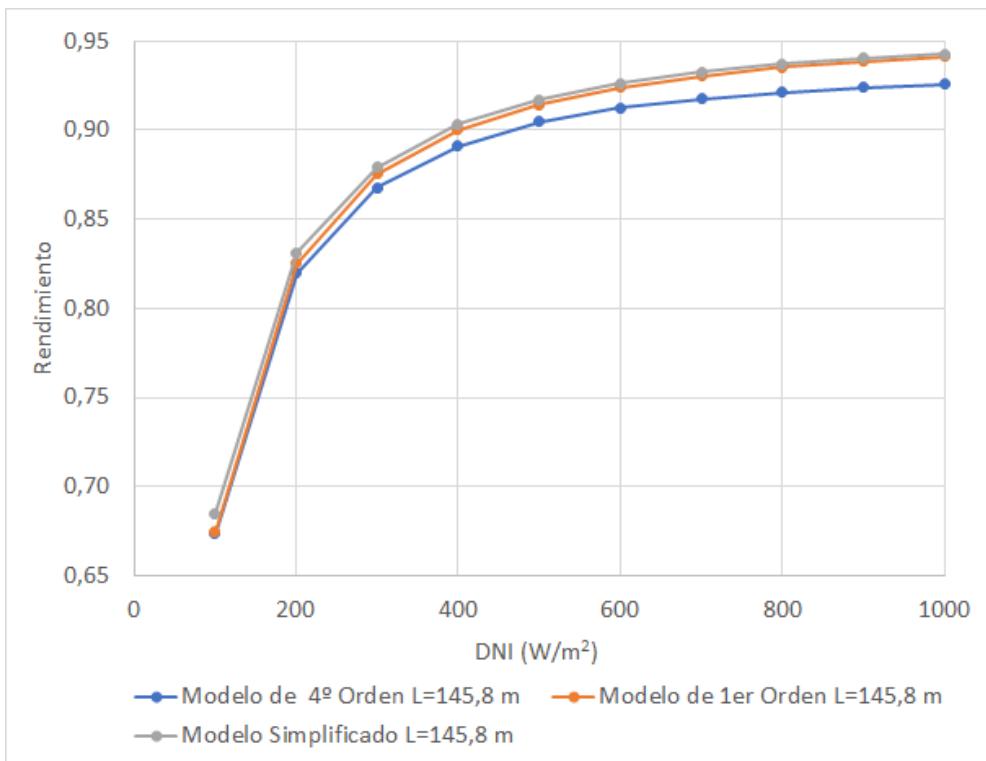


Fig. 4.10. Rendimiento calculado con cada modelo para un tamaño de malla de integración de 145,80 m

En la Fig.4.11 hemos representado el rendimiento, calculado para unas determinadas condiciones de operación, en función del tamaño de malla (el eje de abscisas tiene una escala \log_2). Según aumenta el tamaño de malla se produce una ligera variación en el rendimiento calculado que comienza a crecer más rápidamente alrededor de 100 m, manteniéndose por debajo del 1 % mientras no se supere ese límite de tamaño de malla de integración.

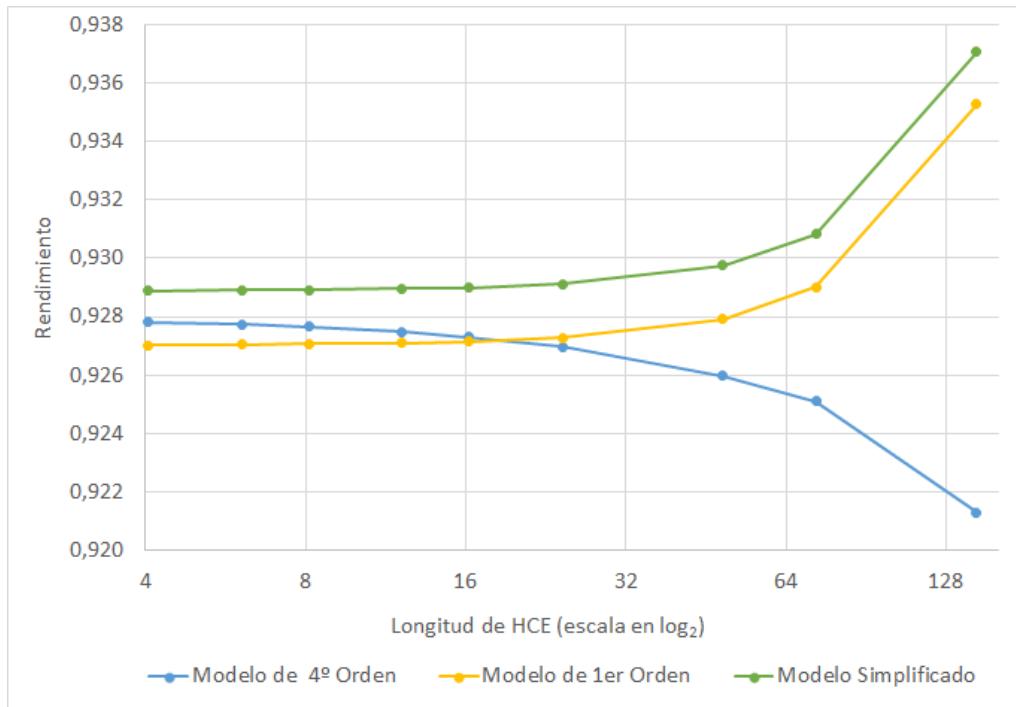


Fig. 4.11. Rendimiento calculado para diferentes tamaños de malla de integración.

$$DNI = 800W/m^2, T_{in} = 300^\circ C, \dot{m} = 6kg/s$$

En la serie de figuras 4.12 a 4.14 podemos ver cómo aumenta la desviación, para cada modelo y en función de DNI , entre el rendimiento calculado para un determinado tamaño de malla y el rendimiento calculado con una malla de 4,05 m (tamaño real del HCE).

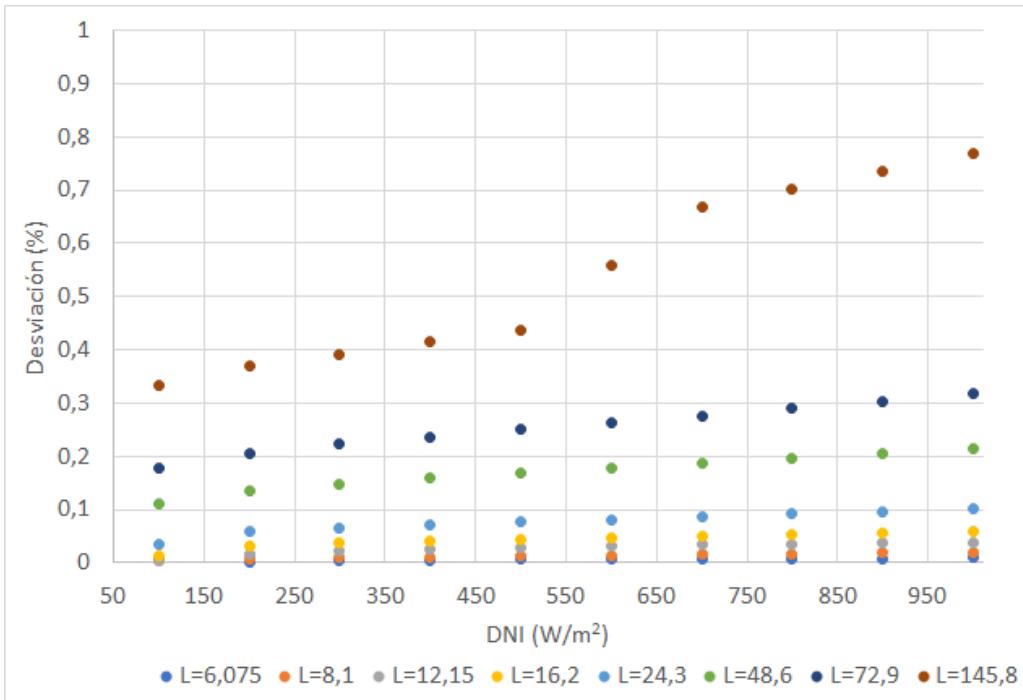


Fig. 4.12. Desviación, para simulaciones con el Modelo de 4º Orden, según diferentes tamaños de malla. Los valores muestran la desviación, en tanto por ciento, respecto a la simulación con una malla de 4,05 m. $T_{in} = 300^\circ\text{C}$, $\dot{m} = 6\text{kg/s}$

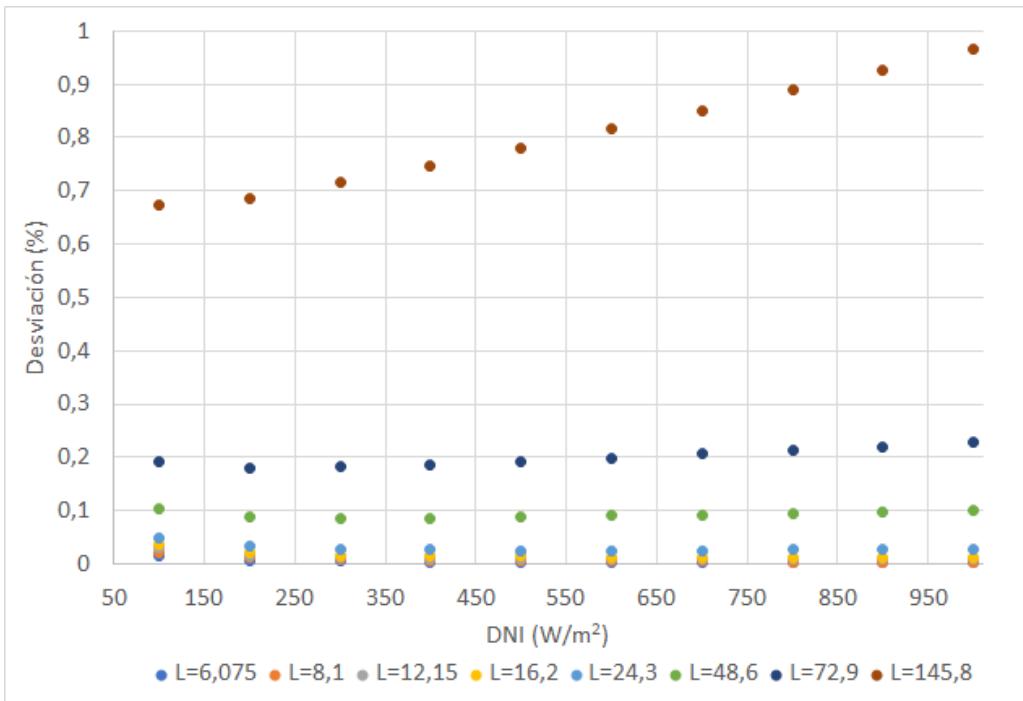


Fig. 4.13. Desviación, para simulaciones con el Modelo de 1^{er} Orden, según diferentes tamaños de malla. Los valores muestran la desviación, en tanto por ciento, respecto a la simulación con una malla de 4,05 m. $T_{in} = 300^\circ\text{C}$, $\dot{m} = 6\text{kg/s}$

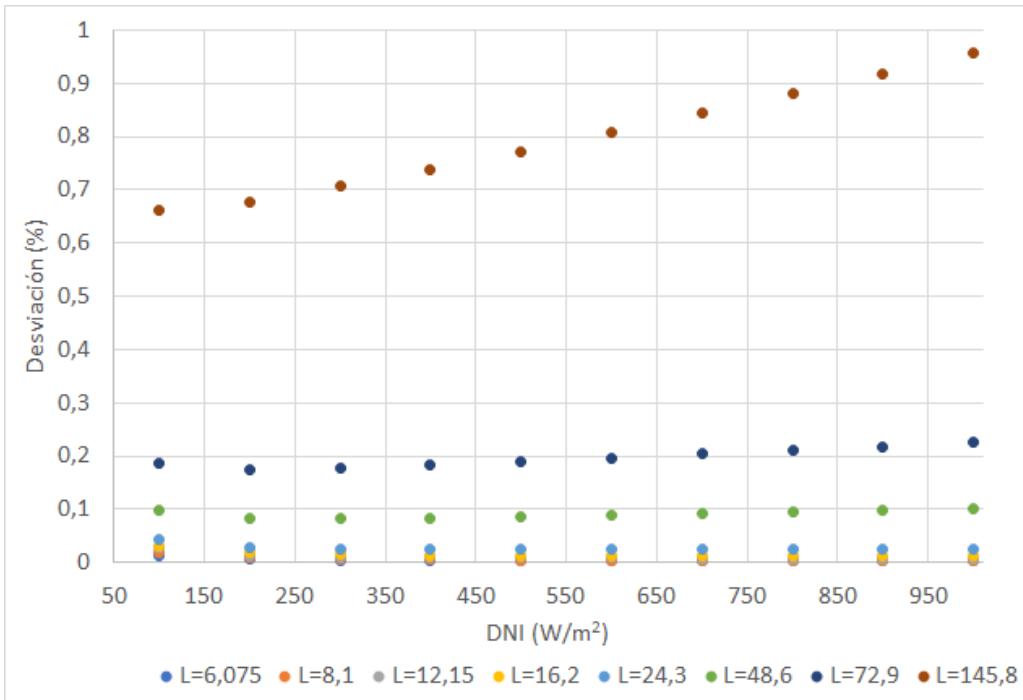


Fig. 4.14. Desviación, para simulaciones con el Modelo de Simplificado, según diferentes tamaños de malla. Los valores muestran la desviación, en tanto por ciento, respecto a la simulación con una malla de 4,05 m. $T_{in} = 300^\circ\text{C}$, $\dot{m} = 6\text{kg/s}$

4.2. Análisis de los datos de generación de una planta solar termoeléctrica real

Finalmente emplearemos nuestro programa de simulación en un análisis del estado del campo solar de una planta termosolar real. Puesto que la simulación de sistemas externos al campo solar queda fuera de nuestro alcance, nos centraremos en los datos referentes al campo solar. En concreto, contrastaremos qué potencia térmica se extrae del campo, independientemente de las causas operativas que condicionasen el funcionamiento de la planta en cada momento. Realizaremos la simulación a partir de datos reales (meteorológicos y de generación) de una central termosolar, comparando el salto térmico real con el simulado.

La configuración de campo solar que se va a utilizar a lo largo de las siguientes simulaciones está basada en las plantas termosolares Aste 1A y 1B, que se encuentran situadas en el término municipal de Alcázar de San Juan, provincia de Ciudad Real. Sus coordenadas geográficas son ($39,1^\circ\text{N}$ $3,16^\circ\text{W}$) y la altitud es de 651 m sobre el nivel del mar.



Fig. 4.15. Centrales Termosolares Aste 1A (izq.) y Aste 1B (der.) Fuente: Google Earth

La potencia eléctrica nominal de cada una de ellas es de 49,9 MW. El proyecto inicial consideraba que las plantas contarían con almacenamiento térmico, el cual se construiría durante una segunda fase que finalmente no se llegó a ejecutar, por lo que en la actualidad solo existe generación durante las horas de sol. Se emplearán los datos de Aste 1B, cuya configuración es la siguiente:

El campo solar cuenta con 120 lazos distribuidos de manera irregular en 4 subcampos. La distancia de separación entre lazos es de 16,25 m.

- Subcampo NO, 31 lazos.
- Subcampo NE, 28 lazos.
- Subcampo SO, 27 lazos.
- Subcampo SE, 34 lazos.

Todos los lazos son idénticos, contando con 4 SCAs cada uno en una configuración tipo *U*. El eje de seguimiento perfectamente plano se encuentra alineado en dirección N-S. Cada SCA cuenta con un total de 336 espejos de vidrio fabricados por Flabeg.

La reflectividad de los espejos puede obtenerse a partir de los registros de mantenimiento de ese año. Llegados a este punto encontramos cierta anomalía en los valores recopilados, tal y como puede apreciarse en la 4.16. Se observa cómo durante los meses de junio y julio el valor de la reflectividad promedio comienza a caer drásticamente hasta valores ligeramente inferiores al 60 %. Los valores van mejorando posteriormente, durante el mes de agosto, hasta alcanzar valores normales a partir de septiembre.

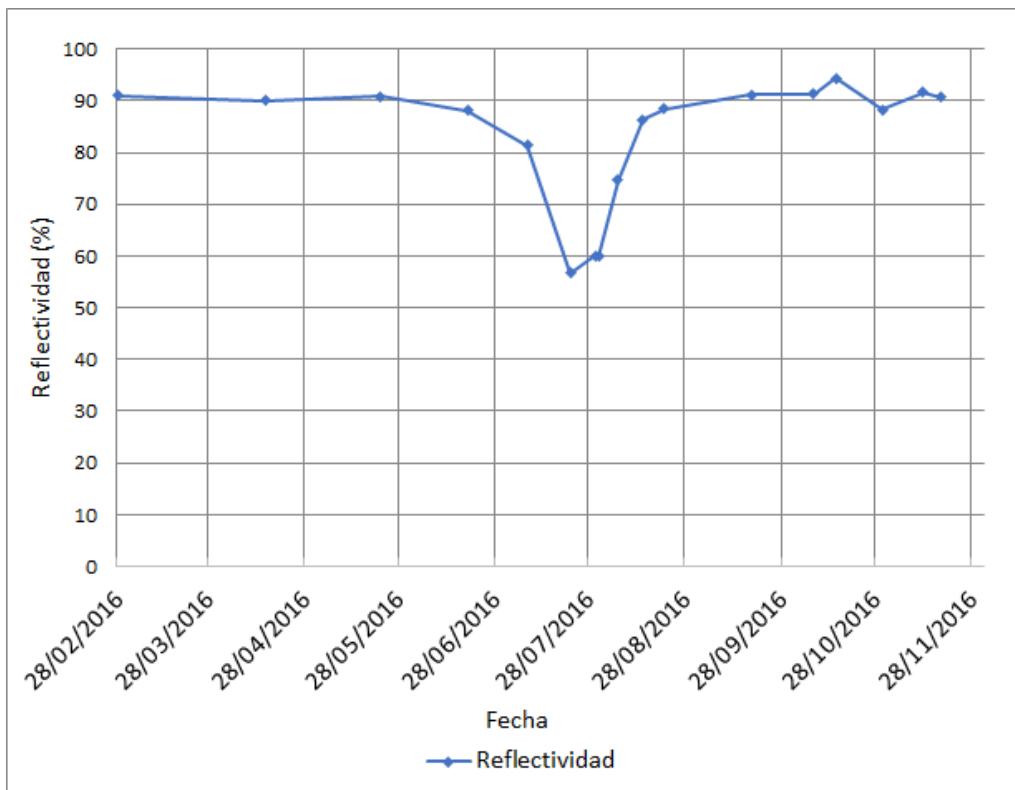


Fig. 4.16. Reflectividad registrada durante el mantenimiento. Fuente: CST Aste 1B

El origen de este comportamiento está en que durante ese periodo se paralizó la actividad de limpieza de espejos (por causas indeterminadas). Una vez que se reanuda el mantenimiento y limpieza del campo solar, los valores de reflectividad vuelven a su valor habitual, alrededor del 90 %. Debe tenerse en cuenta que la limpieza de espejos de todo el campo puede requerir varias semanas para un equipo de mantenimiento compuesto por un solo camión de agua presurizada apoyado por dos operarios con un ritmo normal de limpieza de unos 8 a 10 lazos cada noche. Esta contingencia no afectó a la generación eléctrica de la planta por estar el campo solar muy sobredimensionado.

Respecto al estado de los tubos HCE, consideraremos que se encuentran en buen estado de vacío. Los recuentos anuales efectuados en planta indican que tras 4 años de

operación apenas se contabiliza una media de un HCE sin vacío por cada lazo y el número de HCE sin envolvente de vidrio es despreciable (apenas una veintena de unidades de un total de 17280 unidades en todo el campo).

El fluido de trabajo es *Dowtherm-A*, cuyas propiedades también se han descrito en el apartado 3.2.7.

Los datos meteorológicos son los recogidos a lo largo de 2016 por las tres estaciones meteorológicas con las que cuenta la planta. Al tener por triplicado las medidas de cada variable se adopta el criterio de seleccionar la mediana de las tres y no el valor medio. Esta selección la realiza el sistema de control de planta en cada momento y con este criterio se persigue conseguir una mayor robustez del sistema, pues si una estación presenta valores muy desviados de las otras dos podría darse el caso de que el valor medio estuviese muy alejado del valor verdadero. Cuando por avería o fallo de comunicación se carece de los datos de alguna estación sí se suele adoptar el criterio de seleccionar el valor medio de las dos restantes.

Los datos de las figuras 4.17 a 4.19 muestran el comportamiento de la planta real para un día de buena radiación y condiciones estables. Puesto que la planta no dispone de almacenamiento energético, los efectos de las inercias de arranque y parada se aprecian claramente a primera y última hora del día, así como en los altibajos de radiación en los días inestables. En esta planta, el control de caudal lo realiza manualmente el operador de sala en base a las diferentes limitaciones que el diseño de planta impone. Especialmente importantes son los gradientes de calentamiento o, más apropiadamente, las rampas de calentamiento y enfriamiento que los fabricantes de los equipos recomiendan, como es el caso de la turbina, trenes de generación, bombas principales de HTF o los propios HCEs. También existe una inercia asociada al calentamiento de tuberías y tanques de expansión y rebose. La masa total de HTF en la planta ronda las 1300 toneladas.

En nuestra simulación, la temperatura de salida consignada se alcanza por la mañana más rápidamente de lo que lo hace realmente, al no incluir nuestro modelo ninguna inercia térmica para el calentamiento de todo el sistema. Algo parecido ocurre a la salida, donde el ‘enfriamiento’ de la planta se produce más lentamente en la realidad que en la simulación,
4.17.

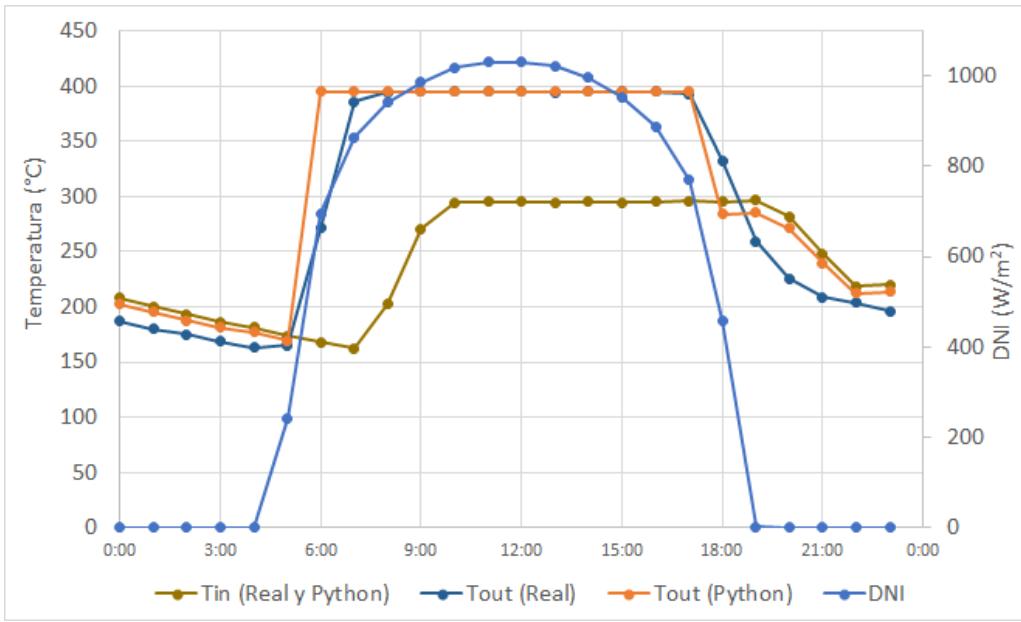


Fig. 4.17. Temperaturas de operación reales y simuladas para el día 1/5/2016. Condiciones estables y buena radiación

En la Fig.4.18 vemos el pico de potencia térmica que realmente se extrae a primera hora de la mañana con el fin de calentar el sistema. En la simulación, esa potencia crece hasta el máximo posible y después se aplana pues, tal y como hemos ido explicando a lo largo de este trabajo, nuestro código simula la operación para un sistema que pudiese extraer toda la energía disponible del campo solar.

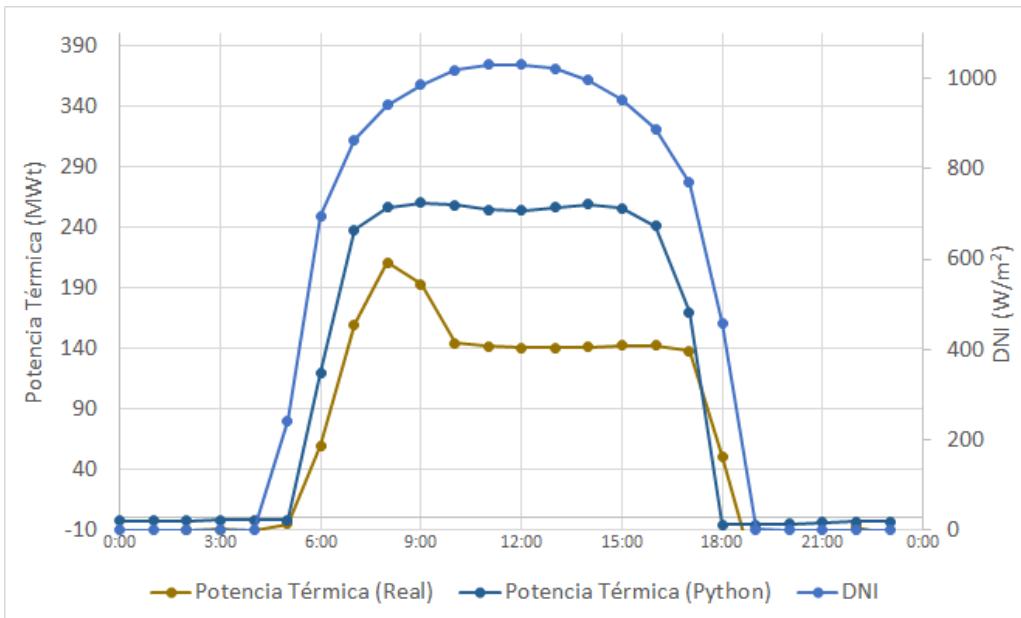


Fig. 4.18. Potencia térmica real y simulada para el día 1/5/2016. Condiciones estables y buena radiación

Pero en la planta real, al no disponer de almacenamiento térmico, una vez que se dispone de suficiente potencia térmica para suministrar al bloque de potencia, el campo debe empezar a limitarse. Es por este motivo por el que vemos en la Fig.4.19 que el caudal simulado durante las horas centrales del día es mucho mayor que el real.

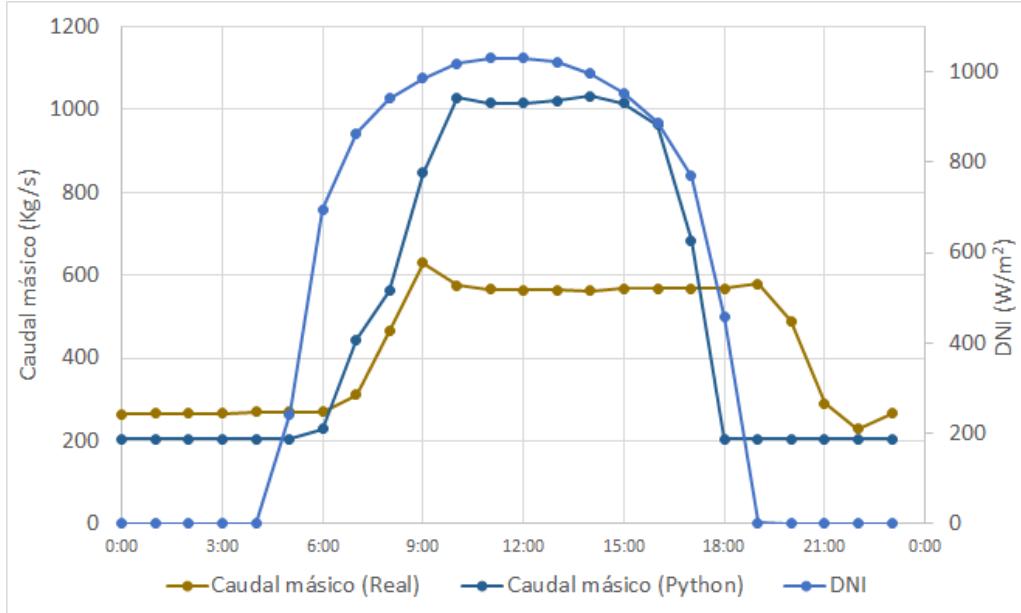


Fig. 4.19. Caudal real y simulado para el día 1/5/2016. Condiciones estables y buena radiación

Aprovecharemos los datos ofrecidos por nuestro código de simulación para estimar el exceso de energía disponible y el rendimiento global del bloque de potencia y de la planta. No pretendemos alcanzar mucha precisión ya que partimos de datos que en algunos casos pueden contener errores de lectura de instrumentos y que no han sido obtenidos de una forma totalmente controlada. Además, se incluyen datos registrados en condiciones muy diferentes de operación, pero no bajo todas las condiciones ni con el mismo peso en el comportamiento global de la planta, por lo que lo se trata de valores promediados entre el conjunto de la muestra de datos.

Para evitar que estos momentos transitorios afecten a nuestros cálculos filtraremos los datos para tener en cuenta solo aquellos momentos en los que las condiciones de generación ya son estables y todo el sistema se encuentra a plena carga. Para ellos seleccionaremos registros en los que la radiación solar directa, DNI, es superior a 700 W/m^2 , la temperatura de entrada y salida del campo están muy cerca de las nominales ($T_{in} > 290^\circ\text{C}$ y $T_{out} > 390^\circ\text{C}$).

Si, bajo este filtro, representamos la potencia térmica real y la potencia térmica simulada (o disponible) en función de DNI, obtenemos la representación de la Fig.4.20. Vemos claramente como la potencia real extraída del campo solar se ha ‘estancado’ alrededor de 140 MWt, mientras que la potencia disponible podría llegar a máximos de algo más de 160 MWt.

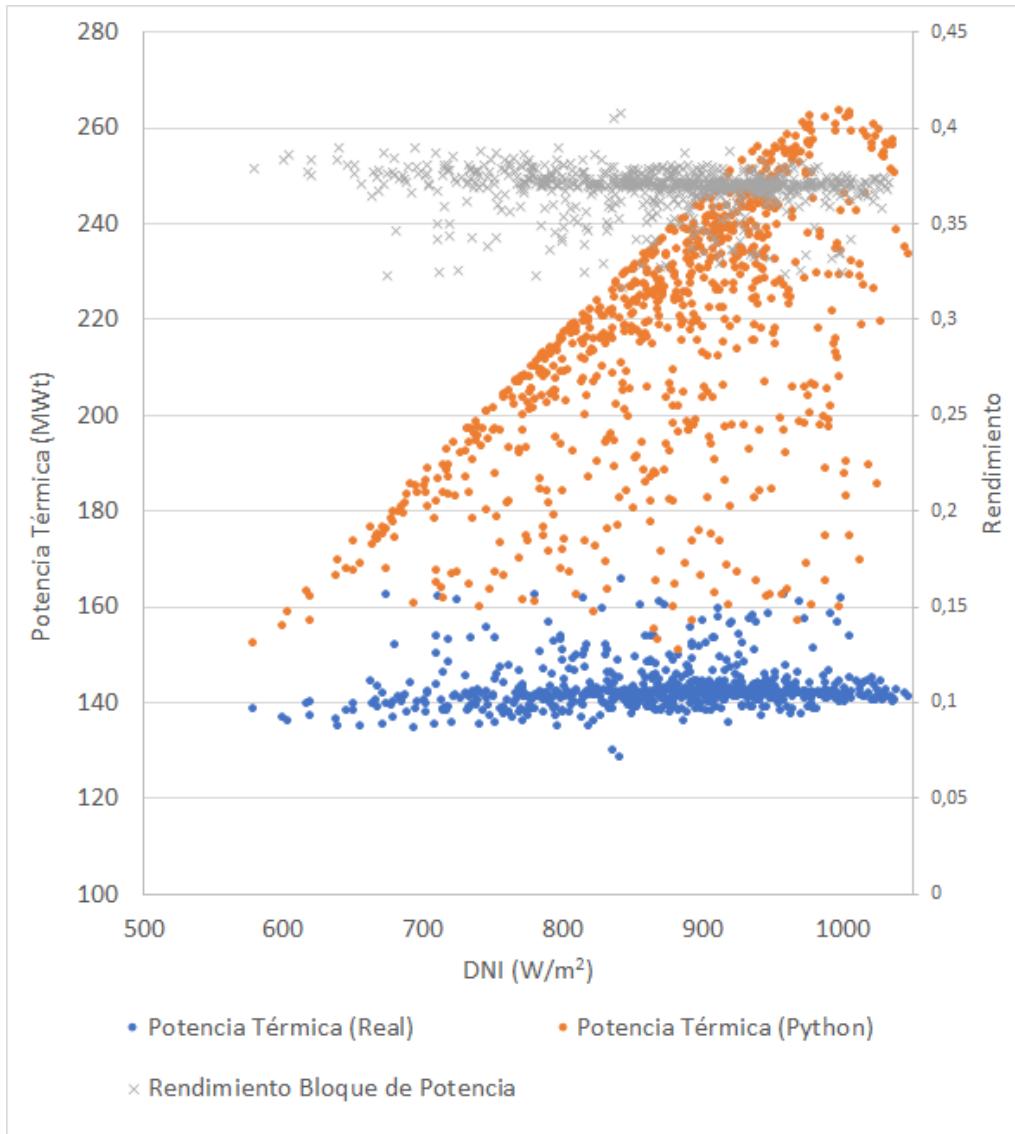


Fig. 4.20. Potencia térmica real y simulada en función de DNI cuando la planta trabaja en condiciones nominales ($T_{in}>290\text{ }^{\circ}\text{C}$, $T_{out}>390\text{ }^{\circ}\text{C}$). En el eje vertical derecho se representa el rendimiento global del bloque de potencia

En esa misma figura se ha representado el rendimiento del bloque de potencia entendido como el cociente de la potencia eléctrica bruta de salida del turbogrupo (52,4 MW) entre la potencia térmica extraída del campo solar (140 MWt). Con esta definición, las

pérdidas en tuberías y el rendimiento de los trenes de generación de vapor queda incluido dentro de este rendimiento global. El valor promedio es aproximadamente 0,37 durante condiciones de generación a plena carga.

Nuestro código nos puede servir, llegado a este punto, para cuantificar qué cantidad de energía solar se está desaprovechando debido al sobredimensionado del campo. En condiciones de diseño ($DNI = 800W/m^2$) el campo solar llega a producir 210 MWt de energía térmica, un 50 % más de la demanda necesaria. Este cálculo es compatible con la experiencia de otras plantas en las que no existe almacenamiento, donde es habitual que el campo solar cuente con unos 96 lazos. En plantas de 50 MW con un almacenamiento de unas 6-8 horas el campo solar ronda los 156 lazos.

5. CONCLUSIONES Y TRABAJO FUTURO

Tal y como se indicaba en el primer capítulo, el objetivo de este trabajo era ganar conocimiento y experiencia para una mejor comprensión del funcionamiento de un campo solar de tecnología CCP y poder así desarrollar herramientas de simulación adecuadas para estos sistemas tan complejos. Hemos podido ver que el desarrollo desde cero de una aplicación para la simulación de un campo solar de CCP requiere sortear un buen número de obstáculos, especialmente si se quiere dotar de cierta versatilidad al código y que éste tenga un carácter general, no exclusivo de unos modelos y configuraciones particulares.

Nuestro principal objetivo se ha cumplido, pues disponemos de un código que nos permite implementar los modelos teóricos de partida y realizar diferentes simulaciones que nos ayudan a comprender el comportamiento de estos sistemas. El código desarrollado permite simular desde un único HCE hasta un campo solar completo, incluyendo cambios en el tipo de fluido caloportador, diferentes opciones de entrada de datos e incluso que cada HCE del campo solar tuviese una configuración diferente.

No obstante, a lo largo del desarrollo, según el número de líneas de código iba creciendo y creciendo, hemos podido comprobar que se requiere un desarrollo mucho más largo y profundo hasta poder alcanzar otras funcionalidades de las que disponen herramientas como SAM, cuya primera versión se remonta a 2007 y sigue siendo desarrollado por parte de un equipo multidisciplinar de científicos. No obstante, pese a las limitaciones funcionales de nuestro código, creemos que son acertadas ciertas bases sobre las que se asienta filosofía, como la Programación Orientación a Objetos que permite su desarrollo modular y también trabajar con los distintos elementos del sistema para realizar otro tipo de simulaciones, no solo de una planta completa de generación de energía térmica y eléctrica.

De cara al futuro, son muchas las mejoras que se podrían realizar, comenzando por una optimización del código, que redujese los tiempos de cálculo. No era nuestro objetivo obtener una herramienta con fines de producción, sino más bien didácticos. Por este motivo, el código tampoco cuenta con una programación *a la defensiva* que evite la introducción de datos inválidos, lo que hubiera multiplicado el número de líneas de código y

horas de trabajo persiguiendo un objetivo que carece de interés científico. Es de resaltar que actualmente la suma de los archivos *interface.py* y *csenergy.py* casi supera la cifra de 5000 de líneas de código, a lo que habría que añadir unos cuantos centenares más en diferentes *scripts* necesarios durante la elaboración de este trabajo.

Pero desde un punto de vista más relacionado con la ingeniería, pensamos que, para el futuro, algunas de las líneas de trabajo más interesantes serían:

- Desarrollar o acoplar nuestro campo solar con un módulo que simule el ciclo de potencia. Someramente, nuestro código facilitaría los datos de caudal y temperatura de HTF disponibles, mientras que el módulo de simulación del ciclo de potencia le devolvería el caudal aprovechado y la temperatura de retorno, más fría, una vez extraída la energía del HTF.
- Si se desarrollan módulos adicionales para la simulación del ciclo de potencia, los consumos auxiliares, bombeos, almacenamiento térmico, etc., sería posible utilizar este código para realizar un estudio del funcionamiento óptimo del campo solar, con el fin de maximizar el rendimiento del conjunto.

Con estas ampliaciones se podría disponer de una herramienta muy versátil, de gran utilidad para el análisis de las diferentes configuraciones del campo solar durante la fase de diseño, ayudando así en el proceso de búsqueda de una solución óptima.

BIBLIOGRAFÍA

- [1] R. Barbero Fresno, “Desarrollo de Un Modelo Teórico Para La Caracterización Del Rendimiento Térmico En Colectores Solares. Aplicación a Tecnologías de Generación Eléctrica,” Tesis doct., Universidad Nacional de Educación a Distancia, 2018.
- [2] *Welcome to Python.Org*, en, <https://www.python.org/>.
- [3] *NumPy*, <https://numpy.org/>.
- [4] *Pandas - Python Data Analysis Library*, <https://pandas.pydata.org/>.
- [5] I. H. Bell, J. Wronski, S. Quoilin y V. Lemort, “Pure and Pseudo-Pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp,” en, *Industrial & Engineering Chemistry Research*, vol. 53, n.º 6, pp. 2498-2508, feb. de 2014. doi: [10.1021/ie4033999](https://doi.org/10.1021/ie4033999).
- [6] W. F. Holmgren, C. W. Hansen y M. A. Mikofski, “Pvlib Python: A Python Package for Modeling Solar Energy Systems,” en, *Journal of Open Source Software*, vol. 3, n.º 29, p. 884, sep. de 2018. doi: [10.21105/joss.00884](https://doi.org/10.21105/joss.00884).
- [7] *TkInter - Python Wiki*, <https://wiki.python.org/moin/TkInter>.
- [8] J. M. Freeman et al., “System Advisor Model (SAM) General Description (Version 2017.9.5),” en, inf. téc. NREL/TP-6A20-70414, 1440404, mayo de 2018, NREL/TP-6A20-70414, 1440404. doi: [10.2172/1440404](https://doi.org/10.2172/1440404).
- [9] M. T. Islam, N. Huda, A. B. Abdullah y R. Saidur, “A Comprehensive Review of State-of-the-Art Concentrating Solar Power (CSP) Technologies: Current Status and Research Trends,” en, *Renewable and Sustainable Energy Reviews*, vol. 91, pp. 987-1018, ago. de 2018. doi: [10.1016/j.rser.2018.04.097](https://doi.org/10.1016/j.rser.2018.04.097).
- [10] *Protermosolar*, es, <https://www.protermosolar.com/>.
- [11] A. Patnode, “Simulation and Performance Evaluation of Parabolic Trough Solar Power Plants,” Mechanical Engineering, University of Wisconsin-Madison, 2006.
- [12] V. E. Dudley et al., “Test Results: SEGS LS-2 Solar Collector,” Sandia National Laboratories, Test Result SAND94-1884, dic. de 1994.

- [13] F Burkholder y C Kutscher, “Heat Loss Testing Of Schott’s 2008 PTR70 Parabolic Trough Receiver,” National Renewable Energy Laboratory, Technical Report NREL/ TP - 550 - 45633, May-2009.
- [14] F. Burkholder y C. Kutscher, “Heat-Loss Testing of Solel’s UVAC3 Parabolic Trough Receiver,” English, National Renewable Energy Lab. (NREL), Golden, CO (United States), inf. téc. NREL/TP-550-42394, ene. de 2008. doi: [10.2172/922153](https://doi.org/10.2172/922153).
- [15] H. Hottel y A. Whillier, “Evaluation of Flat-Plate Solar Collector Performance,” English, *Trans. Conf. Use of Solar Energy; ()*, vol. 3 (Thermal Processes) Part 2, ene. de 1955.
- [16] N. Fraidenraich, J. M Gordon y R. de Cassia Fernandes de Lima, “Improved Solutions for Temperature and Thermal Power Delivery Profiles in Linear Solar Collectors,” en, *Solar Energy*, vol. 61, n.º 3, pp. 141-145, sep. de 1997. doi: [10.1016/S0038-092X\(97\)00049-2](https://doi.org/10.1016/S0038-092X(97)00049-2).
- [17] S. A. Kalogirou, “A Detailed Thermal Model of a Parabolic Trough Collector Receiver,” *Energy*, 2012. doi: [10.1016/j.energy.2012.06.023](https://doi.org/10.1016/j.energy.2012.06.023).
- [18] J. A. Duffie y W. A. Beckman, *Solar Engineering of Thermal Processes*, en, Third. John Wiley & Sons, 2006.
- [19] C. Kutscher, F. Burkholder y J. Kathleen Stynes, “Generation of a Parabolic Trough Collector Efficiency Curve From Separate Measurements of Outdoor Optical Efficiency and Indoor Receiver Heat Loss,” en, *Journal of Solar Energy Engineering*, vol. 134, n.º 1, p. 011 012, feb. de 2012. doi: [10.1115/1.4005247](https://doi.org/10.1115/1.4005247).
- [20] R Forristall, “Heat Transfer Analysis and Modeling of a Parabolic Trough Solar Receiver Implemented in Engineering Equation Solver,” en, inf. téc. NREL/TP-550-34169, 15004820, oct. de 2003, NREL/TP-550-34169, 15004820. doi: [10.2172/15004820](https://doi.org/10.2172/15004820).
- [21] E. Zarza, *Apuntes Del Master Consultor En Energías Renovables*, 2006.
- [22] V. Sharma, J. Nayak y S. Kedare, “Shading and Available Energy in a Parabolic Trough Concentrator Field,” en, *Solar Energy*, vol. 90, pp. 144-153, abr. de 2013. doi: [10.1016/j.solener.2013.01.002](https://doi.org/10.1016/j.solener.2013.01.002).

- [23] P Gilman et al., “Solar Advisor Model User Guide for Version 2.0,” National Renewable Energy Laboratory, Technical Report NREL/TP-670-43704, ago. de 2008.
- [24] Richard L. Moore, “Implementation of DOWTHERM A Properties into RELAP5-3D/ATHENA,” en, inf. téc. INL/EXT-10-18651, 1037788, abr. de 2010, INL/EXT-10-18651, 1037788. doi: [10.2172/1037788](https://doi.org/10.2172/1037788).
- [25] M. Machado, “A Product Technical Data DOWTHERM A Heat Transfer Fluid,” en,
- [26] I Reda y A Andreas, “Solar Position Algorithm for Solar Radiation Applications (Revised),” en, inf. téc. NREL/TP-560-34302, 15003974, ene. de 2008, NREL/TP-560-34302, 15003974. doi: [10.2172/15003974](https://doi.org/10.2172/15003974).

ANEXO A. GLOSARIO

Acrónimos

CCP	Colector Cilindro-Parabólico
CSV	Comma Separated Values
DNI	Direct Normal Irradiance (Radiación Normal Directa, (W/m^2))
HCE	Heat Collector Element
HTF	Heat Transfer Fluid
IAM	Incidence Angle Modifier (Modificador por angulo de incidencia)
JSON	JavaScript Object Notation
IPH	Industrial Process Heat
NREL	National Renewable Energy Laboratory
OOP	Object Oriented Programming
POO	Programación Orientada a Objetos
PTC	Parabolic Trough Collector
SAM	System Advisor Model
SCA	Solar Collector Assembly
SCE	Solar Collector Element
SPA	Solar Position Algorithm
TFG	Trabajo de Fin de Grado
TMY	Typical Meteorological Year
UNED	Universidad Nacional de Educación a Distancia

Símbolos latinos

A_0, A_1	constantes del modelo de emisividad equivalente para receptores, (-)
A_c	superficie de apertura del concentrador, (m^2)
$A_{cs,b}$	sección transversal de la unión entre el soporte y el tubo absorbedor, (m^2)
A_{ext}	área exterior del receptor expuesta a la radiación solar, (m^2)
A_p	Apertura de la superficie parabólica, (m)

C_g	factor de concentración
c_P	calor específico a presión constante, ($J/kg \cdot K$)
D_{gi}	diámetro interior de la envolvente de vidrio del tubo absorbedor (m)
D_{go}	diámetro exterior de la envolvente de vidrio del tubo absorbedor (m)
D_L	distancia de separación entre lazos, (m)
D_{ri}	diámetro interior del tubo absorbedor (m)
D_{ro}	diámetro exterior del tubo absorbedor (m)
F_0, F_1	constantes para el cálculo del IAM según la ec.((3.17)), (-)
F'_{crit}	coeficiente adimensional crítico del modelo segúnlia ec.((2.27))
f_0, f_1, f_2, f_3, f_4	factores adimensionales del modelo según las ecuaciones (2.14), (2.20)-(2.23)
f_l	distancia focal, (m)
$g(Z)$	función característica del rendimiento térmicos según la ec.(2.15), (-)
g', g'', g''', g^{IV}	derivadas de orden 1 a 4 de la función característica $g(Z)$, (-)
$h(T)$	entalpía específica, (J/kg)
\dot{H}	tasa temporal de variación de la entalpía del HTF en el sistema, (J/s)
\bar{h}_b	coeficiente de transmisión por convección medio en el soporte, ($W/(m^2 \cdot K)$)
h_{ext}	coeficiente de transferencia de calor convectivo equivalente
h_{int}	coeficiente de transferencia de calor convectivo hacia el interior, ($W/(m^2 \cdot K)$)
k_b	conductividad térmica del acero del brazo soporte del receptor, ($W/(m \cdot K)$)
k_f	conductividad térmica a la temperatura del fluido, ($W/(m \cdot K)$)
k_{rec}	conductividad del material del receptor, ($W/(m \cdot K)$)
k_T	conductividad térmica, ($W/m \cdot K$)
L	longitud del tubo absorbedor (m)
L_{bordes}	longitud del tubo absorbedor que no recibe radiación, (m)
\dot{m}	caudal másico del HTF, (kg/s)
NTU_{perd}	número de Unidades de Transmisión, (-)
NTU_{perd}	número de Unidades de Transmisión definido según la ec.(2.28), (-)
Nu_G	número de Nusselt calculado con la correlación de Gnielinski, ec.((3.16)), (-)
P_b	perímetro del soporte del receptor, (m)
Pr_f	número de Prandtl a la temperatura del fluido, (-)
Pr_{ri}	número de Prandtl a la temperatura de la pared interior del receptor, (-)
\dot{q}_{abs}	potencia calorífica absorbida a lo largo de todo el HCE, (W)

\dot{q}_{abs}''	flujo de radiación absorbido por la superficie exterior del receptor, (W/m^2)
\dot{q}_{perd}	potencia calorífica perdida a lo largo de todo el HCE, (W)
$\dot{q}_{perd,sopores}$	potencia calorífica perdida a través de los soportes del HCE, (W)
$\dot{q}_{perd}''(x)$	flujo de calor de pérdidas al exterior desde la superficie del receptor, (W/m^2)
$\dot{q}_u''(x)$	flujo de calor útil hacia el interior del receptor, (W/m^2)
T_{base}	temperatura de la zona de conexión entre los brazos y el tubo absorbedor, (K)
T_{ext}	temperatura ambiente exterior, (K)
T_f	temperatura del fluido, (K)
\bar{T}_f	temperatura media del fluido, (K)
$T(h)$	temperatura en función de la entalpía, (K)
T_{in}	temperatura del fluido a la entrada del receptor, (K)
T_{out}	temperatura del fluido a la salida del receptor, (K)
$T_{ro}(x)$	temperatura de pared exterior, (K)
U_{crit}	coeficiente de transmisión de calor al interior crítico según la ec.(2.30), ($W/m^2 \cdot K$)
U_{ext}	coeficiente de transmisión de calor al exterior ($W/m^2 \cdot K$)
U_{rec}	coeficiente global de transferencia de calor hacia el interior ($W/m^2 \cdot K$)
W_{spd}	velocidad del viento, (m/s)
x	coordenada longitudinal, (m)
x^*	coordenada longitudinal adimensional en función de la longitud del tubo, (-)
Z	variable adimensional del modelo de rendimiento térmico según la ec.(2.13), (-)

Símbolos griegos

α	absortividad del receptor, (-)
β	ángulo de seguimiento, (rad)
γ	fracción solar, (-)
γ_L	factor de longitud efectiva, (-)
γ_g	factor de interceptación geométrico, (-)
ε_{ext}	emisividad equivalente de la superficie exterior del tubo, (-)
η_{bordes}	coeficiente de pérdidas geométricas, (-)
$\eta_{disponibilidad}$	factor de disponibilidad, (-)
$\eta_{geométrico}$	factor geométrico, (-)

$\eta_{opt}(\theta)$	rendimiento óptico, (-)
$\eta_{opt,pico}(\theta)$	rendimiento óptico pico, (-)
$\eta_{seguidor}$	factor de precisión del seguidor, (-)
η_{sombra}	coeficiente de pérdidas por sombreado, (-)
$\eta_{suciedad}$	factor de suciedad, (-)
η_T	rendimiento para la totalidad del receptor en el modelo simplificado, (-)
$\eta(x)$	rendimiento integral hasta una distancia x de la entrada, (-)
$\eta_x(x)$	rendimiento local en una sección a distancia x de la entrada, (-)
θ	ángulo de incidencia, (rad)
$\mu(T)$	viscosidad dinámica, ($Pa \cdot m$)
ρ	reflectividad, (-)
$\rho(T)$	densidad (kg/m^3)
σ	constante de Stefan-Boltzmann ($5,67 \times 10^{-8} W/(m^2 \cdot K^4)$)
τ	transmisividad de la cubierta de vidrio del receptor, (-)

ANEXO B. CÓDIGO FUENTE: CSENERGY.PY

```

1 # -*- coding: utf-8 -*-
2
3 """
4 csenergy.py: A Python 3 library for modeling of parabolic-trough solar
5 collectors
6 @author: pacomunuera
7 2020
8 """
9
10 import numpy as np
11 import scipy as sc
12 from CoolProp.CoolProp import PropsSI
13 import CoolProp.CoolProp as CP
14 import pandas as pd
15 import pvlib as pvlib
16 from tkinter import *
17 from tkinter.filedialog import askopenfilename
18 from datetime import datetime, timedelta
19 import os.path
20 import matplotlib.pyplot as plt
21 from matplotlib import rc
22 import json
23
24
25 class Model:
26
27     def calc_pr(self):
28         pass
29
30     def get_hext_eext(self, hce, reext, tro, wind):
31         eext = 0.
32         hext = 0.
33
34         if hce.parameters['Name'] == 'Solel UVAC 2/2008':
35             pass
36
37         elif hce.parameters['Name'] == 'Solel UVAC 3/2010':
38             pass
39
40         elif hce.parameters['Name'] == 'Schott PTR70':
41             pass
42
43         if (hce.parameters['coating'] == 'CERMET' and
44             hce.parameters['annulus'] == 'VACUUM'):
45
46             if wind > 0:
47                 eext = 1.69E-4*reext**0.0395*tro+1/(11.72+3.45E-6*reext)
48                 hext = 0.
49             else:
50                 eext = 2.44E-4*tro+0.0832
51                 hext = 0.
52
53         elif (hce.parameters['coating'] == 'CERMET' and
54               hce.parameters['annulus'] == 'NOVACUUM'):
55             if wind > 0:
56                 eext = ((4.88E-10 * reext**0.0395 + 2.13E-4) * tro +
57                         1 / (-36 - 1.29E-4 * reext) + 0.0962)
58                 hext = 2.34 * reext**0.0646
59             else:
60                 eext = 1.97E-4 * tro + 0.0859
61                 hext = 3.65

```

```

61     elif (hce.parameters['coating'] == 'BLACK CHROME' and
62         hce.parameters['annulus'] == 'VACUUM'):
63         if wind > 0:
64             eext = (2.53E-4 * reext**0.0614 * tro +
65                     1 / (9.92 + 1.5E-5 * reext))
66             hext = 0.
67         else:
68             eext = 4.66E-4 * tro + 0.0903
69             hext = 0.
70     elif (hce.parameters['coating'] == 'BLACK CHROME' and
71         hce.parameters['annulus'] == 'NOVACUUM'):
72         if wind > 0:
73             eext = ((4.33E-10 * reext + 3.46E-4) * tro +
74                     1 / (-20.5 - 6.32E-4 * reext) + 0.149)
75             hext = 2.77 * reext**0.043
76         else:
77             eext = 3.58E-4 * tro + 0.115
78             hext = 3.6
79
80     return hext, eext
81
82
83 class ModelBarbero4thOrder(Model):
84
85     def __init__(self, settings):
86
87         self.parameters = settings
88         self.max_err_t = self.parameters['max_err_t']
89         self.max_err_tro = self.parameters['max_err_tro']
90         self.max_err_pr = self.parameters['max_err_pr']
91
92     def calc_pr(self, hce, htf, row, qabs = None):
93
94         if qabs is None:
95             qabs = hce.qabs
96
97         tin = hce.tin
98
99         # If the hce is the first one in the loop tf = tin, else
100        # tf equals tin plus half the jump of temperature in the previous hce
101
102        if hce.hce_order == 0:
103            tf = hce.tin # HTF bulk temperature
104        else:
105            tf = hce.tin + 0.5 * (hce.sca.hces[hce.hce_order - 1].tout -
106                                  hce.sca.hces[hce.hce_order - 1].tin)
107
108        massflow = hce.sca.loop.massflow
109        wspd = row[1]['Wspd'] # Wind speed
110        text = row[1]['DryBulb'] # Dry bulb ambient temperature
111        sigma = sc.constants.sigma # Stefan-Boltzmann constant
112        dro = hce.parameters['Absorber tube outer diameter']
113        dri = hce.parameters['Absorber tube inner diameter']
114
115        L = (hce.parameters['Length'] * hce.parameters['Bellows ratio'] *
116             hce.parameters['Shield shading'])
117
118        x = 1 # Calculation grid fits hce longitude
119
120        # Specific Capacity

```

```

121     cp = htf.get_specific_heat(tf, hce.pin)
122
123     # Internal transmission coefficient.
124     # hint = hce.get_hint(tf, hce.pin, htf)
125
126     # Internal heat transfer coefficient. Eq. 3.22 Barbero2016
127     urec = hce.get_urec(tf, hce.pin, htf)
128
129     # We suppose thermal performance, pr = 1, at first if the hce is
130     # the first one in the loop or pr_j = pr_j-1 if there is a previous
131     # HCE in the loop.
132     pr = hce.get_previous_pr()
133     tro1 = tf + qabs * pr / urec
134
135     # HCE emittance
136     eext = hce.get_eext(tro1, wspd)
137     # External Convective Heat Transfer equivalent coefficient
138     hext = hce.get_hext(wspd)
139
140     # Thermal power lost through brackets
141     qlost_brackets = hce.get_qlost_brackets(tro1, text)
142
143     # Thermal power lost. Eq. 3.23 Barbero2016
144     # qlost = sigma * eext * (tro1**4 - text**4) + hext * (tro1 - text) + \
145     #     qlost_brackets
146     qlost = sigma * eext * (tro1**4 - text**4) + hext * (tro1 - text)
147
148     # Critical Thermal power loss. Eq. 3.50 Barbero2016
149     qcrit = sigma * eext * (tf**4 - text**4) + hext * (tf - text)
150
151     # Critical Internal heat transfer coefficient, Eq. 3.51 Barbero2016
152     ucrit = 4 * sigma * eext * tf**3 + hext
153
154     # Transmission Units Number, Ec. 3.30 Barbero2016
155     NTU = urec * x * L * np.pi * dro / (massflow * cp)
156
157     if qabs > 1.1 * qcrit:
158
159         # We use Barbero2016's simplified model approximation
160         # Eq. 3.63 Barbero2016
161         fcrit = 1 / (1 + (ucrit / urec))
162
163         # Eq. 3.71 Barbero2016
164         pr = fcrit * (1 - qcrit / qabs)
165
166         errtro = 10.
167         errpr = 1.
168         step = 0
169
170         while ((errtro > self.max_err_tro or errpr > self.max_err_pr) and
171                 step < 1000):
172
173             step += 1
174
175             # Eq. 3.32 Barbero2016
176             f0 = qabs / (urec * (tf - text))
177
178             # Eq. 3.34 Barbero2016
179             f1 = ((4 * sigma * eext * text**3) + hext) / urec
180             f2 = 6 * (text**2) * (sigma * eext / urec) * (qabs / urec)

```

```

181     f3 = 4 * text * (sigma * eext / urec) * ((qabs / urec)**2)
182     f4 = (sigma * eext / urec) * ((qabs / urec)**3)
183
184     pr0 = pr
185
186     fx = lambda pr0: (1 - pr0 -
187                         f1 * (pr0 + (1 / f0)) -
188                         f2 * ((pr0 + (1 / f0))**2) -
189                         f3 * ((pr0 + (1 / f0))**3) -
190                         f4 * ((pr0 + (1 / f0))**4))
191
192     dfx = lambda pr0: (-1 - f1 -
193                         2 * f2 * (pr0 + (1 / f0)) -
194                         3 * f3 * ((pr0 + (1 / f0))**2) -
195                         4 * f4 * ((pr0 + (1 / f0))**3))
196
197     root = sc.optimize.newton(fx,
198                               pr0,
199                               fprime=dfx,
200                               maxiter=100000)
201
202     pr0 = root
203
204     # Eq. 3.37 Barbero2016
205     z = pr0 + (1 / f0)
206
207     # Eq. 3.40, 3.41 & 3.42 Babero2016
208     g1 = 1 + f1 + 2 * f2 * z + 3 * f3 * z**2 + 4 * f4 * z**3
209     g2 = 2 * f2 + 6 * f3 * z + 12 * f4 * z**2
210     g3 = 6 * f3 + 24 * f4 * z
211
212     # Eq. 3.39 Barbero2016
213     pr2 = ((pr0 * g1 / (1 - g1)) * (1 / (NTU * x)) *
214             (np.exp((1 - g1) * NTU * x / g1) - 1) -
215             (g2 / (6 * g1)) * (pr0 * NTU * x)**2 -
216             (g3 / (24 * g1)) * (pr0 * NTU * x)**3))
217
218     errpr = abs(pr2-pr)
219     pr = pr2
220     hce.pr = pr
221
222     hce.set_tout(htf)
223     hce.set_pout(htf)
224     tf = 0.5 * (hce.tin + hce.tout)
225
226     # Specific Capacity
227     cp = htf.get_specific_heat(tf, hce.pin)
228
229     # Internal heat transfer coefficient. Eq. 3.22 Barbero2016
230     urec = hce.get_urec(tf, hce.pin, htf)
231
232     # HCE emittance
233     eext = hce.get_eext(tro1, wspd)
234
235     # External Convective Heat Transfer equivalent coefficient
236     hext = hce.get_hext(wspd)
237
238     tro2 = tf + qabs * pr / urec
239     errtro = abs(tro2-tro1)
240     tro1 = tro2

```

```

241
242     # Increase qlost with thermal power lost through brackets
243     qlost_brackets = hce.get_qlost_brackets(tro1, text)
244
245     # Thermal power loss. Eq. 3.23 Barbero2016
246     qlost = sigma * eext * (tro1**4 - text**4)
247
248     # Critical Thermal power loss. Eq. 3.50 Barbero2016
249     qcrit = sigma * eext * (tf**4 - text**4) + hext * (tf - text)
250
251     # Critical Internal heat transfer coeff. Eq. 3.51 Barbero2016
252     ucrit = 4 * sigma * eext * tf**3 + hext
253
254     # Transmission Units Number, Ec. 3.30 Barbero2016
255     NTU = urec * x * L * np.pi * dro / (massflow * cp)
256
257 if step == 1000:
258     print('No se alcanzó convergencia. HCE', hce.get_index())
259     print(qabs, qcrit, urec, ucrit)
260
261 hce.pr = hce.pr * (1 - qlost_brackets / qabs)
262 hce.qlost = qlost
263 hce.qlost_brackets = qlost_brackets
264
265 else:
266     hce.pr = 0.0
267     errtro = 10.0
268     tf = 0.5 * (hce.tin + hce.tout)
269     tro1 = tf - 5
270     while (errtro > self.max_err_tro):
271
272         kt = htf.get_thermal_conductivity(tro1, hce.pin)
273
274         fx = lambda tro1: ((2 * np.pi * kt * (tf - tro1) /
275                             np.log(dro/dri)) -
276                             sigma * hce.get_eext(tro1, wspd) *
277                             (tro1**4 - text**4) - hce.get_hext(wspd) -
278                             hce.get_qlost_brackets(tro1, text))
279
280         root = sc.optimize.newton(fx,
281                               tro1,
282                               maxiter=100000)
283
284         tro2 = root
285         eext = hce.get_eext(tro2, wspd)
286         # External Convective Heat Transfer equivalent coefficient
287         hext = hce.get_hext(wspd)
288
289         # Thermal power lost. Eq. 3.23 Barbero2016
290         qlost = sigma * eext * (tro2**4 - text**4) + \
291                 hext * (tro2 - text)
292
293         # Thermal power lost through brackets
294         qlost_brackets = hce.get_qlost_brackets(tro2, text)
295
296         hce.qlost = qlost
297         hce.qlost_brackets = qlost_brackets
298         hce.set_tout(htf)
299         hce.set_pout(htf)
300         tf = 0.5 * (hce.tin + hce.tout)

```

```

301         errtro = abs(tro2 - tro1)
302         tro1 = tro2
303
304
305 class ModelBarbero1stOrder(Model):
306
307     def __init__(self, settings):
308
309         self.parameters = settings
310         self.max_err_t = self.parameters['max_err_t']
311         self.max_err_tro = self.parameters['max_err_tro']
312         self.max_err_pr = self.parameters['max_err_pr']
313
314     def calc_pr(self, hce, htf, row, qabs = None):
315
316         if qabs is None:
317             qabs = hce.qabs
318
319         # If the hce is the first one in the loop tf = tin, else
320         # tf equals tin plus half the jump of temperature in the previous hce
321         if hce.hce_order == 0:
322             tf = hce.tin # HTF bulk temperature
323         else:
324             tf = hce.tin + 0.5 * (hce.sca.hces[hce.hce_order - 1].tout -
325                                   hce.sca.hces[hce.hce_order - 1].tin)
326
327         massflow = hce.sca.loop.massflow
328         wspd = row[1]['Wspd'] # Wind speed
329         text = row[1]['DryBulb'] # Dry bulb ambient temperature
330         sigma = sc.constants.sigma # Stefan-Bolztmann constant
331         dro = hce.parameters['Absorber tube outer diameter']
332         dri = hce.parameters['Absorber tube inner diameter']
333         x = 1 # Calculation grid fits hce longitude
334
335         # Specific Capacity
336         cp = htf.get_specific_heat(tf, hce.pin)
337
338         # Internal heat transfer coefficient. Eq. 3.22 Barbero2016
339         urec = hce.get_urec(tf, hce.pin, htf)
340         # We suppose performance, pr = 1, at first
341         pr = 1.0
342         tro1 = tf + qabs * pr / urec
343
344         # HCE emittance
345         eext = hce.get_eext(tro1, wspd)
346         # External Convective Heat Transfer equivalent coefficient
347         hext = hce.get_hext(wspd)
348
349         # Thermal power lost through brackets
350         qlost_brackets = hce.get_qlost_brackets(tro1, text)
351
352         # Thermal power lost. Eq. 3.23 Barbero2016
353         qlost = sigma * eext * (tro1**4 - text**4) + hext * (tro1 - text)
354
355         # Critical Thermal power loss. Eq. 3.50 Barbero2016
356         qcrit = sigma * eext * (tf**4 - text**4) + hext * (tf - text)
357
358         # Critical Internal heat transfer coefficient, Eq. 3.51 Barbero2016
359         ucrit = 4 * sigma * eext * tf**3 + hext
360

```

```

361     # Ec. 3.63
362     fcrit = 1 / (1 + (ucrit / urec))
363
364     # Ec. 3.64
365     Aext = np.pi * dro * x # Pendiente de confirmar
366     NTUpert = ucrit * Aext / (massflow * cp)
367
368     if qabs > qcrit:
369
370         errtro = 10.
371         errpr = 1.
372         step = 0
373
374         while ((errtro > self.max_err_tro or errpr > self.max_err_pr) and
375                step < 1000):
376
377             step += 1
378             pr2 = ((1 - (qcrit / qabs)) *
379                    (1 / (NTUpert * x)) *
380                    (1 - np.exp(-NTUpert * fcrit * x)))
381
382             errpr = abs(pr2-pr)
383             pr = pr2
384             hce.pr = pr
385
386             hce.set_tout(htf)
387             hce.set_pout(htf)
388             tf = 0.5 * (hce.tin + hce.tout)
389
390             # Specific Capacity
391             cp = htf.get_specific_heat(tf, hce.pin)
392
393             # Internal heat transfer coefficient. Eq. 3.22 Barbero2016
394             urec = hce.get_urec(tf, hce.pin, htf)
395
396             # HCE emittance
397             eext = hce.get_eext(tro1, wspd)
398
399             # External Convective Heat Transfer equivalent coefficient
400             hext = hce.get_hext(wspd)
401
402             tro2 = tf + qabs * pr / urec
403             errtro = abs(tro2-tro1)
404             tro1 = tro2
405
406             # Increase qlost with thermal power lost through brackets
407             qlost_brackets = hce.get_qlost_brackets(tro1, text)
408
409             # Thermal power loss. Eq. 3.23 Barbero2016
410             qlost = sigma * eext * (tro1**4 - text**4)
411
412             # Critical Thermal power loss. Eq. 3.50 Barbero2016
413             qcrit = sigma * eext * (tf**4 - text**4) + hext * (tf - text) + \
414                 qlost_brackets
415
416             # Critical Internal heat transfer coefficient, Eq. 3.51 Barbero2016
417             ucrit = 4 * sigma * eext * tf**3 + hext
418
419             # Ec. 3.63
420             fcrit = 1 / (1 + (ucrit / urec))

```

```

421
422     # Transmission Units Number, Ec. 3.30 Barbero2016
423     NTUpert = ucrit * Aext / (massflow * cp)
424
425
426     if step == 1000:
427         print('No se alcanzó convergencia. HCE', hce.get_index())
428         print(qabs, qcrit, urec, ucrit)
429
430         hce.pr = hce.pr * (1 - qlost_brackets / qabs)
431         hce.qlost = qlost
432         hce.qlost_brackets = qlost_brackets
433
434
435     else:
436         hce.pr = 0.0
437         errtro = 10.0
438         tf = 0.5 * (hce.tin + hce.tout)
439         tro1 = tf - 5
440         while (errtro > self.max_err_tro):
441
442             kt = htf.get_thermal_conductivity(tro1, hce.pin)
443
444             fx = lambda tro1: ((2 * np.pi * kt * (tf - tro1) /
445                             np.log(dro/dri)) -
446                             sigma * hce.get_eext(tro1, wspd) *
447                             (tro1**4 - text**4) - hce.get_hext(wspd) -
448                             hce.get_qlost_brackets(tro1, text))
449
450             root = sc.optimize.newton(fx,
451                                     tro1,
452                                     maxiter=100000)
453
454             tro2 = root
455             eext = hce.get_eext(tro2, wspd)
456             # External Convective Heat Transfer equivalent coefficient
457             hext = hce.get_hext(wspd)
458
459             qlost = sigma * eext * (tro2**4 - text**4) + \
460                   hext * (tro2 - text)
461
462             # Thermal power lost through brackets
463             qlost_brackets = hce.get_qlost_brackets(tro2, text)
464
465             hce.qlost = qlost
466             hce.qlost_brackets = qlost_brackets
467             hce.set_tout(htf)
468             hce.set_pout(htf)
469             tf = 0.5 * (hce.tin + hce.tout)
470             errtro = abs(tro2 - tro1)
471             tro1 = tro2
472
473
474 class ModelBarberoSimplified(Model):
475
476     def __init__(self, settings):
477
478         self.parameters = settings
479         self.max_err_t = self.parameters['max_err_t']
480         self.max_err_tro = self.parameters['max_err_tro']

```

```

481         self.max_err_pr = self.parameters['max_err_pr']
482
483     def calc_pr(self, hce, htf, row, qabs=None):
484
485         if qabs is None:
486             qabs = hce.qabs
487
488         # If the hce is the first one in the loop tf = tin, else
489         # tf equals tin plus half the jump of temperature in the previous hce
490         if hce.hce_order == 0:
491             tf = hce.tin # HTF bulk temperature
492         else:
493             tf = hce.tin + 0.5 * (hce.sca.hces[hce.hce_order - 1].tout -
494                                     hce.sca.hces[hce.hce_order - 1].tin)
495
496         wspd = row[1]['Wspd'] # Wind speed
497         text = row[1]['DryBulb'] # Dry bulb ambient temperature
498         sigma = sc.constants.sigma # Stefan-Bolztmann constant
499         dro = hce.parameters['Absorber tube outer diameter']
500         dri = hce.parameters['Absorber tube inner diameter']
501         x = 1 # Calculation grid fits hce longitude
502
503         # Specific Capacity
504         cp = htf.get_specific_heat(tf, hce.pin)
505
506         # Internal heat transfer coefficient. Eq. 3.22 Barbero2016
507         urec = hce.get_urec(tf, hce.pin, htf)
508
509         # We suppose performance, pr = 1, at first
510         pr = 1.0
511         tro1 = tf + qabs * pr / urec
512
513         # HCE emittance
514         eext = hce.get_eext(tro1, wspd)
515         # External Convective Heat Transfer equivalent coefficient
516         hext = hce.get_hext(wspd)
517
518         # Thermal power lost through brackets
519         qlost_brackets = hce.get_qlost_brackets(tro1, text)
520
521         # Thermal power loss. Eq. 3.23 Barbero2016
522         qlost = sigma * eext * (tro1**4 - text**4) + hext * (tro1 - text)
523
524         # Critical Thermal power loss. Eq. 3.50 Barbero2016
525         qcrit = sigma * eext * (tf**4 - text**4) + hext * (tf - text)
526
527         # Critical Internal heat transfer coefficient, Eq. 3.51 Barbero2016
528         ucrit = 4 * sigma * eext * tf**3 + hext
529
530         # Ec. 3.63
531         ## fcrit = (1 / ((4 * eext * tfe**3 / urec) + (hext / urec) + 1))
532         fcrit = 1 / (1 + (ucrit / urec))
533
534         if qabs > qcrit:
535
536             errtro = 10.
537             errpr = 1.
538             step = 0
539
540             while ((errtro > self.max_err_tro or errpr > self.max_err_pr) and

```

```

541         step < 1000):
542
543             step += 1
544             pr2 = fcrit * (1 - (qcrit / qabs))
545
546
547             errpr = abs(pr2-pr)
548             pr = pr2
549             hce.pr = pr
550
551             hce.set_tout(htf)
552             hce.set_pout(htf)
553             tf = 0.5 * (hce.tin + hce.tout)
554
555             # Specific Capacity
556             cp = htf.get_specific_heat(tf, hce.pin)
557
558             # Internal heat transfer coefficient. Eq. 3.22 Barbero2016
559             urec = hce.get_urec(tf, hce.pin, htf)
560
561             # We suppose performance, pr = 1, at first
562             tro1 = tf + qabs * pr / urec
563
564             # HCE emittance
565             eext = hce.get_eext(tro1, wspd)
566
567             # External Convective Heat Transfer equivalent coefficient
568             hext = hce.get_hext(wspd)
569
570             tro2 = tf + qabs * pr / urec
571             errtro = abs(tro2-tro1)
572             tro1 = tro2
573
574             # Thermal power lost through brackets
575             qlost_brackets = hce.get_qlost_brackets(tro1, text)
576
577             # Thermal power loss. Eq. 3.23 Barbero2016
578             qlost = sigma * eext * (tro1**4 - text**4) + hext * (tro1 - text) +
579                         qlost_brackets
580
581             # Critical Thermal power loss. Eq. 3.50 Barbero2016
582             qcrit = sigma * eext * (tf**4 - text**4) + hext * (tf - text)
583
584             # Critical Internal heat transfer coefficient, Eq. 3.51 Barbero2016
585             ucrit = 4 * sigma * eext * tf**3 + hext
586
587             # Ec. 3.63
588             ## fcrit = (1 / ((4 * eext * tfe**3 / urec) + (hext / urec) + 1))
589             fcrit = 1 / (1 + (ucrit / urec))
590
591         if step == 1000:
592             print('No se alcanzó convergencia. HCE', hce.get_index())
593             print(qabs, qcrit, urec, ucrit)
594
595             hce.pr = hce.pr * (1 - qlost_brackets / qabs)
596             hce.qlost = qlost
597             hce.qlost_brackets = qlost_brackets
598
599     else:

```

```

600
601     hce.pr = 0.0
602     errtro = 10.0
603     tf = 0.5 * (hce.tin + hce.tout)
604     tro1 = tf - 5
605     while (errtro > self.max_err_tro):
606
607         kt = htf.get_thermal_conductivity(tro1, hce.pin)
608
609         fx = lambda tro1: ((2 * np.pi * kt * (tf - tro1) /
610                               np.log(dro/dri)) -
611                               sigma * hce.get_eext(tro1, wspd) *
612                               (tro1**4 - text**4) - hce.get_hext(wspd) -
613                               hce.get_qlost_brackets(tro1, text))
614
615         root = sc.optimize.newton(fx,
616                                   tro1,
617                                   maxiter=100000)
618
619         tro2 = root
620         eext = hce.get_eext(tro2, wspd)
621         # External Convective Heat Transfer equivalent coefficient
622         hext = hce.get_hext(wspd)
623
624         qlost = sigma * eext * (tro2**4 - text**4) + \
625                 hext * (tro2 - text)
626
627         # Thermal power lost through brackets
628         qlost_brackets = hce.get_qlost_brackets(tro2, text)
629
630         hce.qlost = qlost
631         hce.qlost_brackets = qlost_brackets
632         hce.set_tout(htf)
633         hce.set_pout(htf)
634         tf = 0.5 * (hce.tin + hce.tout)
635         errtro = abs(tro2 - tro1)
636         tro1 = tro2
637
638         hce.qlost = qlost
639         hce.qlost_brackets = qlost_brackets
640         hce.set_tout(htf)
641         hce.set_pout(htf)
642
643 class HCE(object):
644
645     def __init__(self, sca, hce_order, settings):
646
647         self.sca = sca # SCA in which the HCE is located
648         self.hce_order = hce_order # Relative position of the HCE in the SCA
649         self.parameters = settings # Set of parameters of the HCE
650         self.tin = 0.0 # Temperature of the HTF when enters in the HCE
651         self.tout = 0.0 # Temperature of the HTF when goes out the HCE
652         self.pin = 0.0 # Pressure of the HTF when enters in the HCE
653         self.pout = 0.0 # Pressure of the HTF when goes out the HCE
654         self.pr = 0.0 # Thermal performance of the HCE
655         self.pr_opt = 0.0 # Optical performance of the HCE+SCA set
656         self.qabs = 0.0 # Thermal which reach the absorber tube
657         self.qlost = 0.0 # Thermal power lost through out the HCE
658         self.qlost_brackets = 0.0 # Thermal power lost in brackets and arms
659

```

```

660     def set_tin(self):
661
662         if self.hce_order > 0:
663             self.tin = self.sca.hces[self.hce_order-1].tout
664         elif self.sca.sca_order > 0:
665             self.tin = self.sca.loop.scas[self.sca.sca_order-1].hces[-1].tout
666         else:
667             self.tin = self.sca.loop.tin
668
669     def set_pin(self):
670
671         if self.hce_order > 0:
672             self.pin = self.sca.hces[self.hce_order-1].pout
673         elif self.sca.sca_order > 0:
674             self.pin = self.sca.loop.scas[self.sca.sca_order-1].hces[-1].pout
675         else:
676             self.pin = self.sca.loop.pin
677
678     def set_tout(self, htf):
679
680         if self.pr > 0:
681
682             q = (self.qabs * np.pi * self.pr *
683                  self.parameters['Length'] *
684                  self.parameters['Bellows ratio'] *
685                  self.parameters['Shield shading'] *
686                  self.parameters['Absorber tube outer diameter'])
687
688         else:
689             q = (-1 * (self.qlost) * np.pi *
690                  self.parameters['Length'] *
691                  self.parameters['Bellows ratio'] *
692                  self.parameters['Shield shading'] *
693                  self.parameters['Absorber tube outer diameter'])
694
695         self.tout = htf.get_temperature_by_integration(
696             self.tin, q, self.sca.loop.massflow, self.pin)
697
698
699     def set_pout(self, htf):
700
701         ...
702
703         TO-DO: CÁLCULO PERDIDA DE CARGA:
704
705         Ec. Colebrook-White para el cálculo del factor de fricción de Darcy
706         re_turbulent = 2300
707
708         k = self.parameters['Inner surface roughness']
709         D = self.parameters['Absorber tube inner diameter']
710         re = htf.get_Reynolds(D, self.tin, self.pin, self.sca.loop.massflow)
711
712         if re < re_turbulent:
713             darcy_factor = 64 / re
714
715         else:
716             # a = (k / D) / 3.7
717             a = k / 3.7
718             b = 2.51 / re
719             x = 1

```

```

720         fx = lambda x: x + 2 * np.log10(a + b * x )
721
722         dfx = lambda x: 1 + (2 * b) / (np.log(10) * (a + b * x))
723
724         root = sc.optimize.newton(fx,
725                                     x,
726                                     fprime=dfx,
727                                     maxiter=10000)
728
729         darcy_factor = 1 / (root**2)
730
731         rho = htf.get_density(self.tin, self.pin)
732         v = 4 * self.sca.loop.massflow / (rho * np.pi * D**2)
733         g = sc.constants.g
734
735         # Ec. Darcy-Weisbach para el cálculo de la pérdida de carga
736         deltap_mcl = darcy_factor * (self.parameters['Length'] / D) * \
737             (v**2 / (2 * g))
738         deltap = deltap_mcl * rho * g
739         self.pout = self.pin - deltap
740         ''
741
742         self.pout = self.pin
743
744     def set_pr_opt(self, solarpos):
745
746         IAM = self.sca.get_IAM(solarpos)
747         aoi = self.sca.get_aoi(solarpos)
748         pr_opt_peak = self.get_pr_opt_peak()
749         self.pr_opt = pr_opt_peak * IAM * np.cos(np.radians(aoi))
750
751     def set_qabs(self, aoi, solarpos, row):
752         """Total solar power that reach de absorber tube per longitude unit."""
753         dni = row[1]['DNI']
754         cg = (self.sca.parameters['Aperture'] /
755               (np.pi * self.parameters['Absorber tube outer diameter']))
756
757         pr_shadows = self.get_pr_shadows2(solarpos)
758         pr_borders = self.get_pr_borders(aoi)
759         # Ec. 3.20 Barbero
760         self.qabs = self.pr_opt * cg * dni * pr_borders * pr_shadows
761
762     def get_krec(self, t):
763
764         # From Choom S. Kim for A316
765         # kt = 100*(0.1241+0.00003279*t)
766
767         # Ec. 4.22 Conductividad para el acero 321H.
768         kt = 0.0153 * (t - 273.15) + 14.77
769
770         return kt
771
772     def get urec(self, t, p, htf):
773
774         # HCE wall thermal conductivity
775         krec = self.get_krec(t)
776
777         # Specific Capacity
778         cp = htf.get_specific_heat(t, p)
779

```

```

780     # Internal transmission coefficient.
781     hint = self.get_hint(t, p, htf)
782
783     # Internal heat transfer coefficient. Eq. 3.22 Barbero2016
784     return (1 / ((1 / hint) + (
785         self.parameters['Absorber tube outer diameter'] *
786         np.log(self.parameters['Absorber tube outer diameter'] /
787             self.parameters['Absorber tube inner diameter'])) / 
788         (2 * krec)))
789
790 def get_pr_opt_peak(self):
791
792     alpha = self.get_absorptance()
793     tau = self.get_transmittance()
794     rho = self.sca.parameters['Reflectance']
795     gamma = self.sca.get_solar_fraction()
796
797     pr_opt_peak = alpha * tau * rho * gamma
798
799     if pr_opt_peak > 1 or pr_opt_peak < 0:
800         print("ERROR", pr_opt_peak)
801
802     return pr_opt_peak
803
804 def get_pr_borders(self, aoi):
805
806     if aoi > 90:
807         pr_borders = 0.0
808
809     else:
810         sca_unused_length = (self.sca.parameters["Focal Len"] * 
811                               np.tan(np.radians(aoi)))
812
813         unused_hces = sca_unused_length // self.parameters["Length"]
814
815         unused_part_of_hce = ((sca_unused_length %
816                               self.parameters["Length"]) / 
817                               self.parameters["Length"])
818
819         if self.hce_order < unused_hces:
820             pr_borders = 0.0
821
822         elif self.hce_order == unused_hces:
823             pr_borders = 1 - \
824                 ((sca_unused_length % self.parameters["Length"]) / 
825                   self.parameters["Length"])
826         else:
827             pr_borders = 1.0
828
829         if pr_borders > 1.0 or pr_borders < 0.0:
830             print("ERROR pr_bordes out of limits", pr_borders)
831
832     return pr_borders
833
834 def get_pr_shadows(self, solarpos):
835
836     if solarpos['elevation'][0] < 0:
837         shading = 1
838
839     else:

```

```

840         shading = 1 - (np.sin(np.radians(abs(solarpos['elevation'][0])))) * \
841             self.sca.loop.parameters['row_spacing'] / \
842             self.sca.parameters['Aperture'])
843
844     if shading < 0.0 or shading > 1.0:
845         shading = 0.0
846
847     shading = 1 - shading
848
849     if shading > 1 or shading < 0:
850         print("ERROR shading", shading)
851
852     s2 = self.get_pr_shadows2(solarpos)
853
854     return shading
855
856 def get_pr_shadows2(self, solarpos):
857
858     beta0 = self.sca.get_tracking_angle(solarpos)
859
860     if beta0 >= 0:
861         sigmabeta = 0
862     else:
863         sigmabeta = 1
864
865     # Surface tilt
866     beta = beta0 + 180 * sigmabeta
867
868     surface_azimuth = self.sca.get_surface_azimuth(solarpos)
869
870     Ls = abs(len(self.sca.loop.scas)) * self.sca.parameters['SCA Length'] - \
871         abs(self.sca.loop.parameters['row_spacing']) * \
872             np.tan(np.radians(surface_azimuth - \
873                             solarpos['azimuth'][0])))
874
875     ls = Ls / (len(self.sca.loop.scas) * self.sca.parameters['SCA Length'])
876
877     if solarpos['zenith'][0] < 90:
878         shading = min(abs(np.cos(np.radians(beta0))) * \
879                         self.sca.loop.parameters['row_spacing'] / \
880                         self.sca.parameters['Aperture'], 1)
881     else:
882         shading = 0
883
884     return shading
885
886 def get_hext(self, wspd):
887
888     # TO-DO:
889     return 0.0
890
891 def get_hint(self, t, p, fluid):
892
893     # Gnielinski correlation. Eq. 4.15 Barbero2016
894     kf = fluid.get_thermal_conductivity(t, p)
895     dri = self.parameters['Absorber tube inner diameter']
896
897     # Prandtl number
898     prf = fluid.get_prandtl(t, p)
899

```

```

900     # Reynolds number for absorber tube inner diameter, dri
901     redri = fluid.get_Reynolds(dri, t, p, self.sca.loop.massflow)
902
903     # We suppose inner wall temperature is equal to fluid temperature
904     prri = prf
905
906     # Skin friction coefficient
907     cf = np.power(1.58 * np.log(redri) - 3.28, -2)
908     nug = ((0.5 * cf * prf * (redri - 1000)) /
909             (1 + 12.7 * np.sqrt(0.5 * cf) * (np.power(prf, 2/3) - 1))) * \
910             np.power(prf / prri, 0.11)
911
912     # Internal transmission coefficient.
913     hint = kf * nug / dri
914
915     return hint
916
917 def get_eext(self, tro, wspd):
918
919     # Eq. 5.2 Barbero. Parameters given in Pg. 245
920     eext = (self.parameters['Absorber emittance factor A0'] +
921             self.parameters['Absorber emittance factor A1'] *
922             (tro - 273.15))
923     """
924     If wind speed is lower than 4 m/s, eext is increased up to a 1% at
925     4 m/s. As of 4 m/s forward, eext is increased up to a 2% at 7 m/s
926     """
927     if wspd < 4:
928         eext = eext * (1 + 0.01 * wspd / 4)
929
930     else:
931         eext = eext * (1 + 0.01 * (0.3333 * wspd - 0.3333))
932
933     return eext
934
935 def get_absorptance(self):
936
937     return self.parameters['Absorber absorptance']
938
939 def get_transmittance(self):
940
941     return self.parameters['Envelope transmittance']
942
943 def get_reflectance(self):
944
945     return self.sca.parameters['Reflectance']
946
947 def get_qlost_brackets(self, tf, text):
948
949     # Ec. 4.12
950
951     n = self.parameters['Length'] / self.parameters['Brackets'] + \
952         + (self.hce_order == 0)
953     pb = 0.2032
954     acsb = 1.613e-4
955     kb = 48
956     hb = 20
957     tbase = tf - 10
958
959     L = self.parameters['Length']

```

```

960
961     return n * (np.sqrt(pb * kb * acsb * hb) * (tbase - text)) / L
962
963 def get_previous_pr(self):
964
965     if self.hce_order > 0:
966         previous_pr = self.sca.hces[self.hce_order-1].pr
967     elif self.sca.sca_order > 0:
968         previous_pr = self.sca.loop.scas[self.sca.sca_order-1].hces[-1].pr
969     else:
970         previous_pr = 1.0
971
972     return previous_pr
973
974 def get_index(self):
975
976     if hasattr(self.sca.loop, 'subfield'):
977         index = [self.sca.loop.subfield.name,
978                  self.sca.loop.loop_order,
979                  self.sca.sca_order,
980                  self.hce_order]
981     else:
982         index = ['BL',
983                  self.sca.sca_order,
984                  self.hce_order]
985
986     return index
987
988 class SCA(object):
989
990     def __init__(self, loop, sca_order, settings):
991
992         self.loop = loop
993         self.sca_order = sca_order
994         self.hces = []
995         self.status = 'focused'
996         self.tracking_angle = 0.0
997         self.parameters = dict(settings)
998
999
1000    def get_solar_fraction(self):
1001
1002        if self.status == 'defocused':
1003            solarfraction = 0.0
1004        elif self.status == 'focused':
1005
1006            # Cleanliness two times because it affects mirror and envelope
1007            solarfraction = (self.parameters['Geom.Accuracy'] *
1008                             self.parameters['Track Twist'] *
1009                             self.parameters['Cleanliness'] *
1010                             self.parameters['Cleanliness'] *
1011                             self.parameters['Factor'] *
1012                             self.parameters['Availability'])
1013
1014        else:
1015            solarfraction = 1.0
1016
1017        if solarfraction > 1 or solarfraction < 0:
1018            print("ERROR", solarfraction)
1019

```

```

1020         return solarfraction
1021
1022     def get_IAM(self, solarpos):
1023
1024         if solarpos['zenith'][0] > 80:
1025             kiam = 0.0
1026         else:
1027             aoi = self.get_aoi(solarpos)
1028
1029             F0 = self.parameters['IAM Coefficient F0']
1030             F1 = self.parameters['IAM Coefficient F1']
1031             F2 = self.parameters['IAM Coefficient F2']
1032
1033             if (aoi > 0 and aoi < 80):
1034                 kiam = (F0 + (F1 * np.radians(aoi) + F2 * np.radians(aoi)**2) /
1035                         np.cos(np.radians(aoi)))
1036
1037             if kiam > 1.0:
1038                 kiam = 1.0
1039
1040             if kiam > 1.0 or kiam < 0.0:
1041                 print("ERROR", kiam, aoi)
1042
1043         return kiam
1044
1045     def get_aoi(self, solarpos):
1046
1047         sigmabeta = 0.0
1048         beta0 = 0.0
1049
1050         surface_azimuth = self.get_surface_azimuth(solarpos)
1051         beta0 = self.get_tracking_angle(solarpos)
1052
1053         if beta0 >= 0:
1054             sigmabeta = 0
1055         else:
1056             sigmabeta = 1
1057
1058         beta = beta0 + 180 * sigmabeta
1059         aoi = pvlib.irradiance.aoi(beta,
1060                                     surface_azimuth,
1061                                     solarpos['zenith'][0],
1062                                     solarpos['azimuth'][0])
1063
1064         return aoi
1065
1066     def get_tracking_angle(self, solarpos):
1067
1068         surface_azimuth = self.get_surface_azimuth(solarpos)
1069         # Tracking angle for a collector with tilt = 0
1070         # Ec. 2.32 Technical Manual for the SAM Physical Trough Model
1071         beta0 = np.degrees(
1072             np.arctan(np.tan(np.radians(solarpos['zenith'][0])) *
1073                         np.cos(np.radians(surface_azimuth -
1074                                     solarpos['azimuth'][0]))))
1075
1076         return beta0
1077
1078     def get_surface_azimuth(self, solarpos):
1079
1080         if self.loop.parameters['Tracking Type'] == 1: # N-S single axis
1081             if solarpos['azimuth'][0] > 0 and solarpos['azimuth'][0] <= 180:

```

```

1080         surface_azimuth = 90 # Surface facing east
1081     else:
1082         surface_azimuth = 270 # Surface facing west
1083     elif self.loop.parameters['Tracking Type'] == 2: # E-W single axis
1084         surface_azimuth = 180 # Surface facing the equator
1085
1086     return surface_azimuth
1087
1088
1089 class __Loop__(object):
1090
1091
1092     def __init__(self, settings):
1093
1094         self.scas = []
1095         self.parameters = settings
1096
1097         self.tin = 0.0
1098         self.tout = 0.0
1099         self.pin = 0.0
1100         self.pout = 0.0
1101         self.massflow = 0.0
1102         self.qabs = 0.0
1103         self.qlost = 0.0
1104         self.qlost_brackets = 0.0
1105         self.wasted_power = 0.0
1106         self.pr = 0.0
1107         self.pr_opt = 0.0
1108
1109         self.act_tin = 0.0
1110         self.act_tout = 0.0
1111         self.act_pin = 0.0
1112         self.act_pout = 0.0
1113         self.act_massflow = 0.0 # Actual massflow measured by the flowmeter
1114
1115         self.wasted_power = 0.0
1116         self.tracking = True
1117
1118     def initialize(self, type_of_source, source=None):
1119
1120         if type_of_source == 'rated':
1121             self.massflow = self.parameters['rated_massflow']
1122             self.tin = self.parameters['rated_tin']
1123             self.pin = self.parameters['rated_pin']
1124
1125         elif type_of_source == 'subfield' and source is not None:
1126             self.massflow = source.massflow / len(source.loops)
1127             self.tin = source.tin
1128             self.pin = source.pin
1129
1130         elif type_of_source == 'solarfield' and source is not None:
1131             self.massflow = source.massflow / source.total_loops
1132             self.tin = solarfield.tin
1133             self.pin = solarfield.pin
1134
1135         elif type_of_source == 'values' and source is not None:
1136             self.massflow = source['massflow']
1137             self.tin = source['tin']
1138             self.pin = source['pin']
1139

```

```

1140     else:
1141         print("ERROR initialize()")
1142
1143
1144     def load_actual(self, subfield = None):
1145
1146         if subfield == None:
1147             subfield = self.subfield
1148
1149         self.act_massflow = subfield.act_massflow / len(subfield.loops)
1150         self.act_tin = subfield.act_tin
1151         self.act_tout = subfield.act_tout
1152         self.act_pin = subfield.act_pin
1153         self.act_pout = subfield.act_pout
1154
1155     def set_loop_values_from_HCEs(self):
1156
1157         pr_list = []
1158         qabs_list = []
1159         qlost_brackets_list = []
1160         qlost_list = []
1161         pr_opt_list = []
1162
1163         for s in self.scas:
1164             for h in s.hces:
1165                 pr_list.append(h.pr)
1166                 qabs_list.append(
1167                     h.qabs *
1168                     np.pi *
1169                     h.parameters['Length'] *
1170                     h.parameters['Bellows ratio'] *
1171                     h.parameters['Shield shading'] *
1172                     h.parameters['Absorber tube outer diameter'])
1173                 qlost_brackets_list.append(
1174                     h.qlost_brackets *
1175                     np.pi *
1176                     h.parameters['Length'] *
1177                     h.parameters['Absorber tube outer diameter'])
1178                 qlost_list.append(
1179                     h.qlost *
1180                     np.pi *
1181                     h.parameters['Length'] *
1182                     h.parameters['Absorber tube outer diameter'])
1183                 pr_opt_list.append(h.pr_opt)
1184
1185         self.pr = np.mean(pr_list)
1186         self.qabs = np.sum(qabs_list)
1187         self.qlost_brackets = np.sum(qlost_brackets_list)
1188         self.qlost = np.sum(qlost_list)
1189         self.pr_opt = np.mean(pr_opt_list)
1190
1191     def load_from_base_loop(self, base_loop):
1192
1193         self.massflow = base_loop.massflow
1194         self.tin = base_loop.tin
1195         self.pin = base_loop.pin
1196         self.tout = base_loop.tout
1197         self.pout = base_loop.pout
1198         self.pr = base_loop.pr
1199         self.wasted_power = base_loop.wasted_power

```

```

1200     self.pr_opt = base_loop.pr_opt
1201     self.qabs = base_loop.qabs
1202     self.qlost = base_loop.qlost
1203     self.qlost_brackets = base_loop.qlost_brackets
1204
1205     def calc_loop_pr_for_massflow(self, row, solarpos, htf, model):
1206
1207         for s in self.scas:
1208             aoi = s.get_aoi(solarpos)
1209             for h in s.hces:
1210                 h.set_pr_opt(solarpos)
1211                 h.set_qabs(aoi, solarpos, row)
1212                 h.set_tin()
1213                 h.set_pin()
1214                 model.calc_pr(h, htf, row)
1215
1216         self.tout = self.scas[-1].hces[-1].tout
1217         self.pout = self.scas[-1].hces[-1].pout
1218         self.set_loop_values_from_HCEs()
1219         self.set_wasted_power(htf)
1220
1221     def calc_loop_pr_for_tout(self, row, solarpos, htf, model):
1222
1223         dri = self.scas[0].hces[0].parameters['Absorber tube inner diameter']
1224         min_reynolds = self.scas[0].hces[0].parameters['Min Reynolds']
1225
1226         min_massflow = htf.get_massflow_from_Reynolds(dri, self.tin, self.pin,
1227                                         min_reynolds)
1228
1229         max_error = model.max_err_t # % desviation tolerance
1230         search = True
1231         step = 0
1232
1233         while search:
1234
1235             self.calc_loop_pr_for_massflow(row, solarpos, htf, model)
1236
1237             err = abs(self.tout-self.parameters['rated_tout'])
1238
1239             if err > max_error and step < 1000:
1240                 step += 1
1241                 if self.tout >= self.parameters['rated_tout']:
1242                     self.massflow *= (1 + err / self.parameters['rated_tout'])
1243                     search = True
1244                 elif (self.massflow > min_massflow and
1245                     self.massflow >
1246                     self.parameters['min_massflow']):
1247                     self.massflow *= (1 - err / self.parameters['rated_tout'])
1248                     search = True
1249                 else:
1250                     self.massflow = max(
1251                         min_massflow,
1252                         self.parameters['min_massflow'])
1253                     self.calc_loop_pr_for_massflow(row, solarpos, htf, model)
1254                     search = False
1255             else:
1256                 search = False
1257                 if step>=1000:
1258                     print("Massflow convergence failed")
1259

```

```

1260     self.tout = self.scas[-1].hces[-1].tout
1261     self.pout = self.scas[-1].hces[-1].pout
1262     self.set_loop_values_from_HCEs()
1263     self.wasted_power = 0.0
1264
1265
1266     def check_min_massflow(self, htf):
1267
1268         dri = self.parameters['Absorber tube inner diameter']
1269         t = self.tin
1270         p = self.pin
1271         re = self.parameters['Min Reynolds']
1272         loop_min_massflow = htf.get_massflow_from_Reynolds(
1273             dri, t, p, re)
1274
1275         if self.massflow < loop_min_massflow:
1276             print("Too low massflow", self.massflow, "<",
1277                   loop_min_massflow)
1278
1279     def set_wasted_power(self, htf):
1280
1281         if self.tout > self.parameters['tmax']:
1282             self.wasted_power = self.massflow * htf.get_delta_enthalpy(
1283                 self.parameters['tmax'], self.tout, self.pin, self.pout)
1284         else:
1285             self.wasted_power = 0.0
1286
1287     def get_values(self, type = None):
1288
1289         _values = {}
1290
1291         if type is None:
1292             _values = {'tin': self.tin, 'tout': self.tout,
1293                        'pin': self.pin, 'pout': self.pout,
1294                        'mf': self.massflow}
1295         elif type == 'required':
1296             _values = {'tin': self.tin, 'tout': self.tout,
1297                        'pin': self.pin, 'pout': self.pout,
1298                        'req_mf': self.req_massflow}
1299         elif type == 'actual':
1300             _values = {'actual_tin': self.actual_tin, 'tout': self.tout,
1301                        'pin': self.pin, 'pout': self.pout,
1302                        'mf': self.req_massflow}
1303
1304         return _values
1305
1306     def get_id(self):
1307
1308         return 'LP.{0}.{1:03d}'.format(self.subfield.name, self.loop_order)
1309
1310 class Loop(__Loop__):
1311
1312     def __init__(self, subfield, loop_order, settings):
1313
1314         self.subfield = subfield
1315         self.loop_order = loop_order
1316
1317         super().__init__(settings)
1318
1319

```

```

1320 class BaseLoop(__Loop__):
1321
1322     def __init__(self, settings, sca_settings, hce_settings):
1323
1324         super().__init__(settings)
1325
1326         self.parameters_sca = sca_settings
1327         self.parameters_hce = hce_settings
1328
1329         for s in range(settings['scas']):
1330             self.scas.append(SCA(self, s, sca_settings))
1331             for h in range(settings['hces']):
1332                 self.scas[-1].hces.append(
1333                     HCE(self.scas[-1], h, hce_settings))
1334
1335
1336     def get_id(self, subfield = None):
1337
1338         id = ''
1339         if subfield is not None:
1340             id = 'LB.'+subfield.name
1341         else:
1342             id = 'LB.000'
1343
1344         return id
1345
1346     def get_qlost_brackets(self, tf, text):
1347
1348         # Ec. 4.12
1349
1350         L = self.parameters_hce['Length']
1351         n = (L / self.parameters_hce['Brackets'])
1352         pb = 0.2032
1353         acsb = 1.613e-4
1354         kb = 48
1355         hb = 20
1356         tbase = tf - 10
1357
1358
1359         return n * (np.sqrt(pb * kb * acsb * hb) * (tbase - text)) / L
1360
1361     def get_pr_opt_peak(self):
1362
1363         alpha = self.parameters_hce['Absorber absorptance']
1364         tau = self.parameters_hce['Envelope transmittance']
1365         rho = self.parameters_sca['Reflectance']
1366         gamma = self.get_solar_fraction()
1367
1368         pr_opt_peak = alpha * tau * rho * gamma
1369
1370         if pr_opt_peak > 1 or pr_opt_peak < 0:
1371             print("ERROR pr_opt_peak", pr_opt_peak)
1372
1373
1374     def get_pr_borders(self, aoi):
1375
1376         if aoi > 90:
1377             pr_borders = 0.0
1378
1379         else:

```

```

1380     sca_unused_length = (self.parameters_sca["Focal Len"] *
1381                             np.tan(np.radians(aoi)))
1382
1383     pr_borders = 1 - sca_unused_length / \
1384                 (self.parameters['hces'] * self.parameters_hce["Length"])
1385
1386     if pr_borders > 1.0 or pr_borders < 0.0:
1387         print("ERROR pr_geo out of limits", pr_borders)
1388
1389     return pr_borders
1390
1391
1392 def get_pr_shadows(self, solarpos):
1393
1394     if solarpos['elevation'][0] < 0:
1395         shading = 1
1396
1397     else:
1398         shading = 1 - (np.sin(np.radians(abs(solarpos['elevation'][0])))) *
1399                     self.parameters['row_spacing'] /
1400                     self.parameters_sca['Aperture'])
1401
1402     if shading < 0.0 or shading > 1.0:
1403         shading = 0.0
1404
1405
1406     shading = 1 - shading
1407
1408
1409     if shading > 1 or shading < 0:
1410         print("ERROR shading", shading)
1411
1412     return shading
1413
1414 def get_pr_shadows2(self, solarpos):
1415
1416     beta0 = self.get_tracking_angle(solarpos)
1417
1418     if beta0 >= 0:
1419         sigmabeta = 0
1420     else:
1421         sigmabeta = 1
1422
1423 # Surface tilt
1424     beta = beta0 + 180 * sigmabeta
1425
1426     surface_azimuth = self.get_surface_azimuth(solarpos)
1427
1428     Ls = abs(len(self.scas) * self.parameters_sca['SCA Length'] -
1429              abs(self.parameters['row_spacing']) *
1430                  np.tan(np.radians(surface_azimuth -
1431                                 solarpos['azimuth'][0])))
1432
1433     ls = Ls / (len(self.scas) * self.parameters_sca['SCA Length'])
1434
1435     if solarpos['zenith'][0] < 90:
1436         shading = min(abs(np.cos(np.radians(beta0))) * \
1437                       self.parameters['row_spacing'] / \
1438                       self.parameters_sca['Aperture'], 1)
1439     else:

```

```

1440         shading = 0
1441
1442     return shading
1443
1444
1445 def get_solar_fraction(self):
1446
1447     # Cleanliness two times because it affects mirror and envelope
1448     solarfraction = (self.parameters_sca['Geom.Accuracy'] *
1449                       self.parameters_sca['Track Twist'] *
1450                       self.parameters_sca['Cleanliness'] *
1451                       self.parameters_sca['Cleanliness'] *
1452                       self.parameters_sca['Factor'] *
1453                       self.parameters_sca['Availability'])
1454
1455     if solarfraction > 1 or solarfraction < 0:
1456         print("ERROR", solarfraction)
1457
1458     return solarfraction
1459
1460 def get_IAM(self, solarpos):
1461
1462     if solarpos['zenith'][0] > 80:
1463         kiam = 0.0
1464     else:
1465
1466         aoi = self.get_aoi(solarpos)
1467
1468         F0 = self.parameters_sca['IAM Coefficient F0']
1469         F1 = self.parameters_sca['IAM Coefficient F1']
1470         F2 = self.parameters_sca['IAM Coefficient F2']
1471
1472         if (aoi > 0 and aoi < 80):
1473             kiam = (F0 + (F1 * np.radians(aoi) + F2 * np.radians(aoi)**2) /
1474                     np.cos(np.radians(aoi)))
1475
1476         if kiam > 1.0:
1477             kiam = 1.0
1478
1479         if kiam > 1.0 or kiam < 0.0:
1480             print("ERROR", kiam, aoi)
1481
1482     return kiam
1483
1484
1485 def get_aoi(self, solarpos):
1486
1487     sigmabeta = 0.0
1488     beta0 = 0.0
1489
1490     surface_azimuth = self.get_surface_azimuth(solarpos)
1491     beta0 = self.get_tracking_angle(solarpos)
1492
1493     if beta0 >= 0:
1494         sigmabeta = 0
1495     else:
1496         sigmabeta = 1
1497
1498     beta = beta0 + 180 * sigmabeta
1499     aoi = pvlib.irradiance.aoi(beta,

```

```

1500                     surface_azimuth,
1501                     solarpos['zenith'][0],
1502                     solarpos['azimuth'][0])
1503             return aoi
1504
1505
1506     def get_tracking_angle(self, solarpos):
1507
1508         surface_azimuth = self.get_surface_azimuth(solarpos)
1509         # Tracking angle for a collector with tilt = 0
1510         # Ec. 2.32 Technical Manual for the SAM Physical Trough Model
1511         beta0 = np.degrees(
1512             np.arctan(np.tan(np.radians(solarpos['zenith'][0])) *
1513                 np.cos(np.radians(surface_azimuth -
1514                             solarpos['azimuth'][0]))))
1515     return beta0
1516
1517
1518     def get_surface_azimuth(self, solarpos):
1519
1520         if self.parameters['Tracking Type'] == 1: # N-S single axis tracker
1521             if solarpos['azimuth'][0] > 0 and solarpos['azimuth'][0] <= 180:
1522                 surface_azimuth = 90 # Surface facing east
1523             else:
1524                 surface_azimuth = 270 # Surface facing west
1525         elif self.parameters['Tracking Type'] == 2: # E-W single axis tracker
1526             surface_azimuth = 180 # Surface facing the equator
1527
1528     return surface_azimuth
1529
1530
1531 class Subfield(object):
1532     ''
1533     Parabolic Trough Solar Field
1534     ''
1535
1536     def __init__(self, solarfield, settings):
1537
1538         self.solarfield = solarfield
1539         self.name = settings['name']
1540         self.parameters = settings
1541         self.loops = []
1542
1543         self.tin = 0.0
1544         self.tout = 0.0
1545         self.pin = 0.0
1546         self.pout = 0.0
1547         self.massflow = 0.0
1548         self.qabs = 0.0
1549         self.qlost = 0.0
1550         self.qlost_brackets = 0.0
1551         self.wasted_power = 0.0
1552         self.pr = 0.0
1553         self.pr_opt = 0.0
1554
1555         self.act_tin = 0.0
1556         self.act_tout = 0.0
1557         self.act_pin = 0.0
1558         self.act_pout = 0.0
1559         self.act_massflow = 0.0

```

```

1560     self.pr_act_massflow = 0.0
1561
1562     self.rated_tin = self.solarfield.rated_tin
1563     self.rated_tout = self.solarfield.rated_tout
1564     self.rated_pin = self.solarfield.rated_pin
1565     self.rated_pout = self.solarfield.rated_pout
1566     self.rated_massflow = (self.solarfield.rated_massflow *
1567                             self.parameters['loops'] /
1568                             self.solarfield.total_loops)
1569
1570 def set_subfield_values_from_loops(self, htf):
1571
1572     self.massflow = np.sum([l.massflow for l in self.loops])
1573     self.pr = np.sum([l.pr * l.massflow for l in self.loops]) / \
1574         self.massflow
1575     self.pr_opt = np.sum([l.pr_opt * l.massflow for l in self.loops]) / \
1576         self.massflow
1577     self.wasted_power = np.sum([l.wasted_power for l in self.loops]) / \
1578         1000000 # From Watts to MW
1579     self.pout = np.sum([l.pout * l.massflow for l in self.loops]) / \
1580         self.massflow
1581     self.tout = htf.get_temperature(
1582         np.sum([l.massflow *
1583                 htf.get_enthalpy(l.tout, l.pout) for l in self.loops]) / \
1584                 self.massflow, self.pout)
1585     self.qlost = np.sum([l.qlost for l in self.loops]) / 1000000
1586     self.qabs = np.sum([l.qabs for l in self.loops]) / 1000000
1587     self.qlost_brackets = np.sum([l.qlost_brackets for l in self.loops]) \
1588         / 1000000
1589
1590 def set_massflow(self):
1591
1592     self.massflow = np.sum([l.massflow for l in self.loops])
1593
1594 def set_req_massflow(self):
1595
1596     self.req_massflow = np.sum([l.req_massflow for l in self.loops])
1597
1598 def set_wasted_power(self):
1599
1600     self.wasted_power = np.sum([l.wasted_power for l in self.loops]) / \
1601         1000000 # From Watts to MW
1602
1603 def set_pr_req_massflow(self):
1604
1605     self.pr_req_massflow = np.sum(
1606         [l.pr_req_massflow * l.req_massflow for l in self.loops]) / \
1607         self.req_massflow
1608
1609 def set_pr_act_massflow(self):
1610
1611     self.pr_act_massflow = np.sum(
1612         [l.pr_act_massflow * l.act_massflow for l in self.loops]) / \
1613         self.act_massflow
1614
1615 def set_pout(self):
1616
1617     self.pout = np.sum([l.pout * l.massflow for l in self.loops]) / \
1618         self.massflow
1619

```

```

1620     def set_tout(self, htf):
1621         ...
1622         Calculates HTF output temperature throughout the solar field as a
1623         weighted average based on the enthalpy of the mass flow in each
1624         loop of the solar field
1625         ...
1626         self.tout = htf.get_temperature(
1627             np.sum([l.massflow * htf.get_enthalpy(l.tout, l.pout)
1628                     for l in self.loops]) / self.massflow, self.pout)
1629
1630     def initialize(self, source, values = None):
1631
1632         if source == 'rated':
1633             self.massflow = self.rated_massflow
1634             self.tin = self.rated_tin
1635             self.pin = self.rated_pin
1636             self.tout = self.rated_tout
1637             self.pout = self.rated_pout
1638
1639         elif source == 'actual':
1640             self.massflow = self.act_massflow
1641             self.tin = self.act_tin
1642             self.pin = self.act_pin
1643             self.tout = self.act_tout
1644             self.pout = self.act_pout
1645
1646         elif source == 'values' and values is not None:
1647             self.massflow = values['massflow']
1648             self.tin = values['tin']
1649             self.pin = values['pin']
1650
1651         else:
1652             print('Select source [rated|actual|values]')
1653             sys.exit()
1654
1655     def load_actual(self, row):
1656
1657         self.act_massflow = row[1][self.get_id() +'.a.mf']
1658         self.act_tin = row[1][self.get_id() +'.a.tin']
1659         self.act_pin = row[1][self.get_id() +'.a.pin']
1660         self.act_tout = row[1][self.get_id() +'.a.tout']
1661         self.act_pout = row[1][self.get_id() +'.a.pout']
1662
1663     def get_id(self):
1664
1665         return 'SB.' + self.name
1666
1667 class SolarField(object):
1668
1669     def __init__(self, subfield_settings, loop_settings, sca_settings,
1670                  hce_settings):
1671
1672         self.subfields = []
1673         self.total_loops = 0
1674
1675         self.tin = 0.0
1676         self.tout = 0.0
1677         self.pin = 0.0
1678         self.pout = 0.0
1679         self.massflow = 0.0

```

```

1680     self.qabs = 0.0
1681     self.qlost = 0.0
1682     self.qlost_brackets = 0.0
1683     self.wasted_power = 0.0
1684     self.pr = 0.0
1685     self.pr_opt = 0.0
1686     self.pwr = 0.0
1687
1688     self.act_tin = 0.0
1689     self.act_tout = 0.0
1690     self.act_pin = 0.0
1691     self.act_pout = 0.0
1692     self.act_massflow = 0.0
1693     self.act_pwr = 0.0
1694
1695     self.rated_tin = loop_settings['rated_tin']
1696     self.rated_tout = loop_settings['rated_tout']
1697     self.rated_pin = loop_settings['rated_pin']
1698     self.rated_pout = loop_settings['rated_pout']
1699     self.rated_massflow = (loop_settings['rated_massflow'] *
1700                           self.total_loops)
1701
1702     for sf in subfield_settings:
1703         self.total_loops += sf['loops']
1704         self.subfields.append(Subfield(self, sf))
1705         for l in range(sf['loops']):
1706             self.subfields[-1].loops.append(
1707                 Loop(self.subfields[-1], l, loop_settings))
1708             for s in range(loop_settings['scas']):
1709                 self.subfields[-1].loops[-1].scas.append(
1710                     SCA(self.subfields[-1].loops[-1], s, sca_settings))
1711                 for h in range (loop_settings['hces']):
1712                     self.subfields[-1].loops[-1].scas[-1].hces.append(
1713                         HCE(self.subfields[-1].loops[-1].scas[-1], h,
1714                             hce_settings))
1715
1716     # TO-DO: FUTURE WORK
1717     self.storage_available = False
1718     self.operation_mode = "subfield_heating"
1719
1720 def initialize(self, source, values = None):
1721
1722     if source == 'rated':
1723         self.massflow = self.rated_massflow
1724         self.tin = self.rated_tin
1725         self.pin = self.rated_pin
1726         self.tout = self.rated_tout
1727         self.pout = self.rated_pout
1728
1729     elif source == 'actual':
1730         self.massflow = self.act_massflow
1731         self.tin = self.act_tin
1732         self.pin = self.act_pin
1733         self.tout = self.act_tout
1734         self.pout = self.act_pout
1735
1736     elif source == 'values' and values is not None:
1737         self.massflow = values['massflow']
1738         self.tin = values['tin']
1739         self.pin = values['pin']

```

```

1740
1741     else:
1742         print('Select source [rated|actual|values]')
1743         sys.exit()
1744
1745     def load_actual(self, htf):
1746
1747         self.act_massflow = np.sum([sb.act_massflow for sb in self.subfields])
1748         self.act_pin = np.sum(
1749             [sb.act_pin * sb.act_massflow for sb in self.subfields]) / \
1750             self.act_massflow
1751         self.act_pout = np.sum(
1752             [sb.act_pout * sb.act_massflow for sb in self.subfields]) / \
1753             self.act_massflow
1754         self.act_tin = htf.get_temperature(
1755             np.sum([sb.act_massflow *
1756                 htf.get_enthalpy(sb.act_tin, sb.act_pin)
1757                 for sb in self.subfields]) / self.act_massflow,
1758                 self.act_pin)
1759         self.act_tout = htf.get_temperature(
1760             np.sum([sb.act_massflow *
1761                 htf.get_enthalpy(sb.act_tout, sb.act_pout)
1762                 for sb in self.subfields] / self.act_massflow),
1763                 self.act_pout)
1764
1765     def set_solarfield_values_from_subfields(self, htf):
1766
1767         self.massflow = np.sum([sb.massflow for sb in self.subfields])
1768         self.pr = np.sum(
1769             [sb.pr * sb.massflow for sb in self.subfields]) / self.massflow
1770         self.pr_opt = np.sum(
1771             [sb.pr_opt * sb.massflow for sb in self.subfields]) / self.massflow
1772         self.wasted_power = np.sum([sb.wasted_power for sb in self.subfields])
1773         self.pout = np.sum(
1774             [sb.pout * sb.massflow for sb in self.subfields]) / self.massflow
1775         self.tout = htf.get_temperature(
1776             np.sum([sb.massflow * htf.get_enthalpy(sb.tout, sb.pout)
1777                 for sb in self.subfields]) / self.massflow, self.pout)
1778         self.qlost = np.sum([sb.qlost for sb in self.subfields])
1779         self.qabs = np.sum([sb.qabs for sb in self.subfields])
1780         self.qlost_brackets = np.sum(
1781             [sb.qlost_brackets for sb in self.subfields])
1782
1783     def set_massflow(self):
1784
1785         self.massflow = np.sum([sb.massflow for sb in self.subfields])
1786         self.req_massflow = np.sum([sb.req_massflow for sb in self.subfields])
1787
1788     def set_req_massflow(self):
1789
1790         self.req_massflow = np.sum([sb.req_massflow for sb in self.subfields])
1791
1792     def set_wasted_power(self):
1793
1794         self.wasted_power = np.sum([sb.wasted_power for sb in self.subfields])
1795
1796     def set_pr_req_massflow(self):
1797
1798         self.pr_req_massflow = np.sum(
1799             [sb.pr_req_massflow * sb.req_massflow for sb in self.subfields]) \

```

```

1800     / self.req_massflow
1801
1802     def set_pr_act_massflow(self):
1803
1804         self.pr_act_massflow = np.sum(
1805             [sb.pr_act_massflow * sb.act_massflow for sb in self.subfields]) \
1806             / self.act_massflow
1807
1808     def set_pout(self):
1809
1810         self.pout = np.sum(
1811             [sb.pout * sb.massflow for sb in self.subfields]) \
1812             / self.massflow
1813
1814     def set_tout(self, htf):
1815         """
1816             Calculates HTF output temperature throughout the solar plant as a
1817             weighted average based on the enthalpy of the mass flow in each
1818             subfield of the solar field
1819         """
1820
1821         self.tout = htf.get_temperature(
1822             np.sum([sb.massflow * htf.get_enthalpy(sb.tout, sb.pout) for sb in
1823                 self.subfields]) / self.massflow, self.pout)
1824
1825     def set_act_tout(self, htf):
1826         """
1827             Calculates HTF output temperature throughout the solar plant as a
1828             weighted average based on the enthalpy of the mass flow in each
1829             loop of the solar field
1830         """
1831
1832         self.act_tout = htf.get_temperature(
1833             np.sum([sb.act_massflow *
1834                 tf.get_enthalpy(sb.act_tout, sb.act_pout) for sb in
1835                 self.subfields]) / self.act_massflow, self.act_pout)
1836
1837     def set_tin(self, htf):
1838         """
1839             Calculates HTF output temperature throughout the solar plant as a
1840             weighted average based on the enthalpy of the mass flow in each
1841             loop of the solar field
1842         """
1843
1844         self.tin = tf.get_temperature(
1845             np.sum([sb.massflow *
1846                 tf.get_enthalpy(sb.tin, sb.pin) for sb in
1847                 self.subfields]) / self.massflow, self.pin)
1848
1849     def set_pin(self):
1850
1851         self.pin = np.sum([sb.pin * sb.massflow for sb in self.subfields]) \
1852             / self.massflow
1853
1854     def set_act_pin(self):
1855
1856         self.act_pin = np.sum(
1857             [sb.act_pin * sb.act_massflow for sb in self.subfields]) \
1858             / self.massflow
1859
1860     def set_thermal_power(self, htf, datatype):
1861
1862         self.pwr = self.massflow * \

```

```

1860         htf.get_delta_enthalpy(self.tin, self.tout, self.pin, self.pout)
1861
1862     # From watts to MW
1863     self.pwr /= 1000000
1864
1865     if datatype == 2:
1866         self.act_pwr = self.act_massflow * \
1867             htf.get_delta_enthalpy(
1868                 self.act_tin, self.act_tout, self.act_pin, self.act_pout)
1869
1870     # From watts to MW
1871     self.act_pwr /= 1000000
1872
1873 def print(self):
1874
1875     for sb in self.subfields:
1876         for l in sb.loops:
1877             for s in l.scas:
1878                 for h in s.hces:
1879                     print("subfield: ", sb.name,
1880                           "Lazo: ", l.loop_order,
1881                           "SCA: ", s.sca_order,
1882                           "HCE: ", h.hce_order,
1883                           "tin", "=", h.tin,
1884                           "tout", "=", h.tout)
1885
1886
1887 class SolarFieldSimulation(object):
1888     """
1889     Definimos la clase simulacion para representar las diferentes
1890     pruebas que lancemos, variando el archivo TMY, la configuracion del
1891     site, la planta, el modo de operacion o el modelo empleado.
1892     """
1893
1894     def __init__(self, settings):
1895
1896         self.ID = settings['simulation']['ID']
1897         self.simulation = settings['simulation']['simulation']
1898         self.benchmark = settings['simulation']['benchmark']
1899         self.datatype = settings['simulation']['datatype']
1900         self.fastmode = settings['simulation']['fastmode']
1901         self.tracking = True
1902         self.solarfield = None
1903         self.htf = None
1904         self.coldfluid = None
1905         self.site = None
1906         self.datasource = None
1907         self.powercycle = None
1908         self.parameters = settings
1909         self.first_date = pd.to_datetime(settings['simulation']['first_date'])
1910         self.last_date = pd.to_datetime(settings['simulation']['last_date'])
1911         self.report_df = pd.DataFrame()
1912
1913         if settings['model']['name'] == 'Barbero4thOrder':
1914             self.model = ModelBarbero4thOrder(settings['model'])
1915         elif settings['model']['name'] == 'Barbero1stOrder':
1916             self.model = ModelBarbero1stOrder(settings['model'])
1917         elif settings['model']['name'] == 'BarberoSimplified':
1918             self.model = ModelBarberoSimplified(settings['model'])
1919

```

```

1920     if self.datatype == 1:
1921         self.datasource = Weather(settings['simulation'])
1922     elif self.datatype == 2:
1923         self.datasource = FieldData(settings['simulation'],
1924                                     settings['tags'])
1925
1926     if not hasattr(self.datasource, 'site'):
1927         self.site = Site(settings['site'])
1928     else:
1929         self.site = Site(self.datasource.site_to_dict())
1930
1931
1932     if settings['HTF']['source'] == "CoolProp":
1933         if settings['HTF']['CoolPropID'] not in Fluid._COOLPROP_FLUIDS:
1934             print("Not CoolPropID valid")
1935             sys.exit()
1936         else:
1937             self.htf = FluidCoolProp(settings['HTF'])
1938
1939     else:
1940         self.htf = FluidTabular(settings['HTF'])
1941
1942     self.solarfield = SolarField(settings['subfields'],
1943                                 settings['loop'],
1944                                 settings['SCA'],
1945                                 settings['HCE'])
1946
1947     self.base_loop = BaseLoop(settings['loop'],
1948                               settings['SCA'],
1949                               settings['HCE'])
1950
1951 def runSimulation(self):
1952
1953     self.show_message()
1954
1955     for row in self.datasource.dataframe.iterrows():
1956
1957         if self.datatype == 1: # Because tmy format include TZ info
1958             naive_datetime = datetime.strptime(
1959                 row[0].strftime('%Y/%m/%d %H:%M'), "%Y/%m/%d %H:%M")
1960         else:
1961             naive_datetime = row[0]
1962
1963         if (naive_datetime < self.first_date or
1964             naive_datetime > self.last_date):
1965             pass
1966
1967         else:
1968
1969             solarpos = self.site.get_solarposition(row)
1970
1971             self.gather_general_data(row, solarpos)
1972
1973
1974             if solarpos['zenith'][0] < 90:
1975                 self.tracking = True
1976             else:
1977                 self.tracking = False
1978
1979             if self.simulation:

```

```

1980     self.simulate_solarfield(solarpos, row)
1981     self.solarfield.set_thermal_power(self.htf, self.datatype)
1982     self.gather_simulation_data(row)
1983
1984     str_data = ("SIMULATION: {0} " +
1985                 "DNI: {1:3.0f} W/m2 Qm: {2:4.1f}kg/s " +
1986                 "Tin: {3:4.1f}° Tout: {4:4.1f}°C")
1987
1988     print(str_data.format(row[0],
1989                           row[1]['DNI'],
1990                           self.solarfield.massflow,
1991                           self.solarfield.tin - 273.15,
1992                           self.solarfield.tout - 273.15))
1993
1994
1995     if self.benchmark and self.datatype == 2: # 2: Field Data File
available
1996     self.benchmark_solarfield(solarpos, row)
1997     self.solarfield.set_thermal_power(self.htf, self.datatype)
1998     self.gather_benchmark_data(row)
1999
2000     str_data = ("BENCHMARK: {0} " +
2001                 "DNI: {1:3.0f} W/m2 act_Qm: {2:4.1f}kg/s " +
2002                 "act_Tin: {3:4.1f}° act_Tout: {4:4.1f}° " +
2003                 "Tout: {5:4.1f}°")
2004
2005     print(str_data.format(row[0],
2006                           row[1]['DNI'],
2007                           self.solarfield.act_massflow,
2008                           self.solarfield.act_tin - 273.15,
2009                           self.solarfield.act_tout - 273.15,
2010                           self.solarfield.tout - 273.15))
2011
2012     self.save_results()
2013
2014 def simulate_solarfield(self, solarpos, row):
2015
2016     if self.datatype == 1:
2017         for s in self.solarfield.subfields:
2018             s.initialize('rated')
2019             for l in s.loops:
2020                 l.initialize('rated')
2021         self.solarfield.initialize('rated')
2022         self.base_loop.initialize('rated')
2023     if self.fastmode:
2024         if solarpos['zenith'][0] > 90:
2025             self.base_loop.massflow = \
2026                 self.base_loop.parameters['min_massflow']
2027             self.base_loop.calc_loop_pr_for_massflow(
2028                 row, solarpos, self.htf, self.model)
2029         else:
2030             self.base_loop.calc_loop_pr_for_tout(
2031                 row, solarpos, self.htf, self.model)
2032         for s in self.solarfield.subfields:
2033             for l in s.loops:
2034                 l.load_from_base_loop(self.base_loop)
2035     else:
2036         for s in self.solarfield.subfields:
2037             for l in s.loops:
2038                 if solarpos['zenith'][0] > 90:

```

```

2039         l.massflow = \
2040             self.base_loop.parameters['min_massflow']
2041         l.calc_loop_pr_for_massflow(
2042             row, solarpos, self.htf, self.model)
2043     else:
2044         if l.loop_order > 1:
2045             # For a faster convergence
2046             l.massflow = \
2047                 l.subfield.loops[l.loop_order - 1].massflow
2048         l.calc_loop_pr_for_tout(
2049             row, solarpos, self.htf, self.model)
2050
2051 elif self.datatype == 2:
2052     # 1st, we initialize subfields because actual data are given for
2053     # subfields. 2nd, we initialize solarfield.
2054     for s in self.solarfield.subfields:
2055         s.load_actual(row)
2056         s.initialize('actual')
2057     self.solarfield.load_actual(self.htf)
2058     self.solarfield.initialize('actual')
2059     if self.fastmode:
2060         # Force minimum massflow at night
2061         for s in self.solarfield.subfields:
2062             self.base_loop.initialize('subfield', s)
2063             if solarpos['zenith'][0] > 90:
2064                 self.base_loop.massflow = \
2065                     self.base_loop.parameters['min_massflow']
2066                 self.base_loop.calc_loop_pr_for_massflow(
2067                     row, solarpos, self.htf, self.model)
2068             else:
2069                 self.base_loop.calc_loop_pr_for_tout(
2070                     row, solarpos, self.htf, self.model)
2071         for l in s.loops:
2072             l.load_from_base_loop(self.base_loop)
2073     else:
2074         for s in self.solarfield.subfields:
2075             for l in s.loops:
2076                 # l.load_actual(s)
2077                 l.initialize('subfield', s)
2078                 if solarpos['zenith'][0] > 90:
2079                     l.massflow = l.parameters['min_massflow']
2080                     l.calc_loop_pr_for_massflow(
2081                         row, solarpos, self.htf, self.model)
2082                 else:
2083                     if l.loop_order > 1:
2084                         # For a faster convergence
2085                         l.massflow = \
2086                             l.subfield.loops[l.loop_order - 1].massflow
2087                         l.calc_loop_pr_for_tout(
2088                             row, solarpos, self.htf, self.model)
2089
2090     for s in self.solarfield.subfields:
2091         s.set_subfield_values_from_loops(self.htf)
2092
2093     self.solarfield.set_solarfield_values_from_subfields(self.htf)
2094
2095 def benchmark_solarfield(self, solarpos, row):
2096
2097     for s in self.solarfield.subfields:
2098         s.load_actual(row)

```

```

2099         s.initialize('actual')
2100     self.solarfield.load_actual(self.htf)
2101     self.solarfield.initialize('actual')
2102
2103     if self.fastmode:
2104         for s in self.solarfield.subfields:
2105             self.base_loop.initialize('subfield', s)
2106             self.base_loop.calc_loop_pr_for_massflow(
2107                 row, solarpos, self.htf, self.model)
2108             self.base_loop.set_loop_values_from_HCEs()
2109
2110         for l in s.loops:
2111             l.load_from_base_loop(self.base_loop)
2112
2113             s.set_subfield_values_from_loops(self.htf)
2114
2115     else:
2116         for s in self.solarfield.subfields:
2117             for l in s.loops:
2118                 # l.load_actual()
2119                 l.initialize('subfield', s)
2120                 l.calc_loop_pr_for_massflow(
2121                     row, solarpos, self.htf, self.model)
2122                 l.set_loop_values_from_HCEs()
2123
2124             s.set_subfield_values_from_loops(self.htf)
2125
2126     self.solarfield.set_solarfield_values_from_subfields(self.htf)
2127
2128
2129 def store_values(self, row, values):
2130
2131     for v in values:
2132         self.datasource.dataframe.at[row[0], v] = values[v]
2133
2134 def gather_general_data(self, row, solarpos):
2135
2136     self.datasource.dataframe.at[row[0], 'elevation'] = \
2137         solarpos['elevation'][0]
2138     self.datasource.dataframe.at[row[0], 'zenith'] = \
2139         solarpos['zenith'][0]
2140     self.datasource.dataframe.at[row[0], 'azimuth'] = \
2141         solarpos['azimuth'][0]
2142     aoi = self.base_loop.get_aoi(solarpos)
2143     self.datasource.dataframe.at[row[0], 'aoi'] = aoi
2144     self.datasource.dataframe.at[row[0], 'IAM'] = \
2145         self.base_loop.get_IAM(solarpos)
2146     self.datasource.dataframe.at[row[0], 'pr_shadows'] = \
2147         self.base_loop.get_pr_shadows2(solarpos)
2148     self.datasource.dataframe.at[row[0], 'pr_borders'] = \
2149         self.base_loop.get_pr_borders(aoi)
2150     self.datasource.dataframe.at[row[0], 'pr_opt_peak'] = \
2151         self.base_loop.get_pr_opt_peak()
2152     self.datasource.dataframe.at[row[0], 'solar_fraction'] = \
2153         self.base_loop.get_solar_fraction()
2154
2155 def gather_simulation_data(self, row):
2156
2157     # Solarfield data
2158     self.datasource.dataframe.at[row[0], 'SF.x.mf'] = \

```

```

2159         self.solarfield.massflow
2160     self.datasource.dataframe.at[row[0], 'SF.x.tin'] = \
2161         self.solarfield.tin - 273.15
2162     self.datasource.dataframe.at[row[0], 'SF.x.tout'] = \
2163         self.solarfield.tout - 273.15
2164     self.datasource.dataframe.at[row[0], 'SF.x.pin'] = \
2165         self.solarfield.pin
2166     self.datasource.dataframe.at[row[0], 'SF.x.pout'] = \
2167         self.solarfield.pout
2168     self.datasource.dataframe.at[row[0], 'SF.x.prth'] = \
2169         self.solarfield.pr
2170     self.datasource.dataframe.at[row[0], 'SF.x.prop'] = \
2171         self.solarfield.pr_opt
2172     self.datasource.dataframe.at[row[0], 'SF.x.qabs'] = \
2173         self.solarfield.qabs
2174     self.datasource.dataframe.at[row[0], 'SF.x qlst'] = \
2175         self.solarfield.qlost
2176     self.datasource.dataframe.at[row[0], 'SF.x qlbk'] = \
2177         self.solarfield.qlost_brackets
2178     self.datasource.dataframe.at[row[0], 'SF.x.pwr'] = \
2179         self.solarfield.pwr
2180
2181     if self.datatype == 2:
2182         if row[1]['GrossPower']>0:
2183             self.datasource.dataframe.at[row[0], 'SF.x.globalpr'] = \
2184                 row[1]['GrossPower'] / self.solarfield.pwr
2185         else:
2186             self.datasource.dataframe.at[row[0], 'SF.x.globalpr'] = 0
2187     else:
2188         self.datasource.dataframe.at[row[0], 'SF.x.globalpr'] = 0
2189
2190     if self.fastmode:
2191
2192         values = {
2193             self.base_loop.get_id() + '.x.mf': self.base_loop.massflow,
2194             self.base_loop.get_id() + '.x.tin':
2195                 self.base_loop.tin - 273.15,
2196             self.base_loop.get_id() + '.x.tout':
2197                 self.base_loop.tout - 273.15,
2198             self.base_loop.get_id() + '.x.pin': self.base_loop.pin,
2199             self.base_loop.get_id() + '.x.pout': self.base_loop.pout,
2200             self.base_loop.get_id() + '.x.prth': self.base_loop.pr,
2201             self.base_loop.get_id() + '.x.prop': self.base_loop.pr_opt,
2202             self.base_loop.get_id() + '.x.qabs': self.base_loop.qabs,
2203             self.base_loop.get_id() + '.x qlst': self.base_loop.qlost,
2204             self.base_loop.get_id() + '.x qlbk': \
2205                 self.base_loop.qlost_brackets}
2206         self.store_values(row, values)
2207
2208     for s in self.solarfield.subfields:
2209         # Agregate data from subfields
2210         values = {
2211             s.get_id() + '.x.mf': s.massflow,
2212             s.get_id() + '.x.tin': s.tin - 273.15,
2213             s.get_id() + '.x.tout': s.tout - 273.15,
2214             s.get_id() + '.x.pin': s.pin,
2215             s.get_id() + '.x.pout': s.pout,
2216             s.get_id() + '.x.prth': s.pr,
2217             s.get_id() + '.x.prop': s.pr_opt,
2218             s.get_id() + '.x.qabs': s.qabs,

```

```

2219         s.get_id() + '.x.qlst': s.qlost,
2220         s.get_id() + '.x qlbk': s.qlost_brackets}
2221
2222     self.store_values(row, values)
2223
2224     if not self.fastmode:
2225         for l in s.loops:
2226             # Loop data
2227             values = {
2228                 l.get_id() + '.x.mf': l.massflow,
2229                 l.get_id() + '.x.tin': l.tin - 273.15,
2230                 l.get_id() + '.x.tout': l.tout - 273.15,
2231                 l.get_id() + '.x.pin': l.pin,
2232                 l.get_id() + '.x.pout': l.pout,
2233                 l.get_id() + '.x.prth': l.pr,
2234                 l.get_id() + '.x.prop': l.pr_opt,
2235                 l.get_id() + '.x.qabs': l.qabs,
2236                 l.get_id() + '.x.qlost': l.qlost,
2237                 l.get_id() + '.x qlbk': l.qlost_brackets}
2238
2239             self.store_values(row, values)
2240
2241 def gather_benchmark_data(self, row):
2242
2243     # Solarfield data
2244     self.datasource.dataframe.at[row[0], 'SF.a.mf'] = \
2245         self.solarfield.massflow
2246     self.datasource.dataframe.at[row[0], 'SF.a.tin'] = \
2247         self.solarfield.tin - 273.15
2248     self.datasource.dataframe.at[row[0], 'SF.a.tout'] = \
2249         self.solarfield.act_tout - 273.15
2250     self.datasource.dataframe.at[row[0], 'SF.a.pwr'] = \
2251         self.solarfield.act_pwr
2252     self.datasource.dataframe.at[row[0], 'SF.b.tout'] = \
2253         self.solarfield.tout - 273.15
2254     self.datasource.dataframe.at[row[0], 'SF.b.prth'] = \
2255         self.solarfield.pr
2256     self.datasource.dataframe.at[row[0], 'SF.b.prop'] = \
2257         self.solarfield.pr_opt
2258     self.datasource.dataframe.at[row[0], 'SF.b.pwr'] = \
2259         self.solarfield.pwr
2260     self.datasource.dataframe.at[row[0], 'SF.b.wpwr'] = \
2261         self.solarfield.wasted_power
2262     self.datasource.dataframe.at[row[0], 'SF.a.pin'] = \
2263         self.solarfield.pin
2264     self.datasource.dataframe.at[row[0], 'SF.a.pout'] = \
2265         self.solarfield.act_pout
2266     self.datasource.dataframe.at[row[0], 'SF.b.pout'] = \
2267         self.solarfield.pout
2268     self.datasource.dataframe.at[row[0], 'SF.b.qabs'] = \
2269         self.solarfield.qabs
2270     self.datasource.dataframe.at[row[0], 'SF.b.qlost'] = \
2271         self.solarfield.qlost
2272     self.datasource.dataframe.at[row[0], 'SF.b qlbk'] = \
2273         self.solarfield.qlost_brackets
2274
2275     if self.solarfield.qabs > 0:
2276         self.datasource.dataframe.at[row[0], 'SF.a.prth'] = \
2277             self.solarfield.act_pwr / self.solarfield.qabs
2278 else:

```

```

2279         self.datasource.dataframe.at[row[0], 'SF.a.prth'] = 0
2280
2281     if row[1]['GrossPower']>0:
2282         self.datasource.dataframe.at[row[0], 'SF.a.globalpr'] = \
2283             row[1]['GrossPower'] / self.solarfield.act_pwr
2284     else:
2285         self.datasource.dataframe.at[row[0], 'SF.a.globalpr'] = 0
2286
2287     if row[1]['GrossPower']>0:
2288         self.datasource.dataframe.at[row[0], 'SF.b.globalpr'] = \
2289             row[1]['GrossPower'] / self.solarfield.pwr
2290     else:
2291         self.datasource.dataframe.at[row[0], 'SF.b.globalpr'] = 0
2292
2293     for s in self.solarfield.subfields:
2294
2295         if self.fastmode:
2296             values = {
2297                 self.base_loop.get_id(s) + '.a.mf':
2298                     self.base_loop.massflow,
2299                 self.base_loop.get_id(s) + '.a.tin':
2300                     self.base_loop.tin - 273.15,
2301                 self.base_loop.get_id(s) + '.a.tout':
2302                     self.base_loop.act_tout - 273.15,
2303                 self.base_loop.get_id(s) + '.b.tout':
2304                     self.base_loop.tout - 273.15,
2305                 self.base_loop.get_id(s) + '.a.pin': self.base_loop.pin,
2306                 self.base_loop.get_id(s) + '.b.pout': self.base_loop.pout,
2307                 self.base_loop.get_id(s) + '.b.prth': self.base_loop.pr,
2308                 self.base_loop.get_id(s) + '.b.prop':
2309                     self.base_loop.pr_opt,
2310                 self.base_loop.get_id(s) + '.b.qabs': self.base_loop.qabs,
2311                 self.base_loop.get_id(s) + '.b qlst': self.base_loop.qlost,
2312                 self.base_loop.get_id(s) + '.b qlbk':
2313                     self.base_loop.qlost_brackets,
2314                 self.base_loop.get_id(s) + '.b.wpwr':
2315                     self.base_loop.wasted_power}
2316             self.store_values(row, values)
2317
2318     # Agregate data from subfields
2319     values = {
2320         s.get_id() + '.a.mf': s.act_massflow,
2321         s.get_id() + '.a.tin': s.act_tin - 273.15,
2322         s.get_id() + '.a.tout': s.act_tout - 273.15,
2323         s.get_id() + '.b.tout': s.tout - 273.15,
2324         s.get_id() + '.a.pin': s.act_pin,
2325         s.get_id() + '.a.pout': s.act_pout,
2326         s.get_id() + '.b.pout': s.pout,
2327         s.get_id() + '.b.prth': s.pr,
2328         s.get_id() + '.b.prop': s.pr_opt,
2329         s.get_id() + '.b.qabs': s.qabs,
2330         s.get_id() + '.b qlst': s.qlost,
2331         s.get_id() + '.b qlbk': s.qlost_brackets,
2332         s.get_id() + '.b.wpwr': s.wasted_power}
2333
2334     self.store_values(row, values)
2335
2336     if not self.fastmode:
2337         for l in s.loops:
2338             # Loop data

```

```

2339     values = {
2340         l.get_id() + '.a.mf': l.act_massflow,
2341         l.get_id() + '.a.tin': l.act_tin - 273.15,
2342         l.get_id() + '.a.tout': l.act_tout - 273.15,
2343         l.get_id() + '.b.tout': l.tout - 273.15,
2344         l.get_id() + '.a.pin': l.act_pin,
2345         l.get_id() + '.a.pout': l.act_pout,
2346         l.get_id() + '.b.pout': l.pout,
2347         l.get_id() + '.b.prth': l.pr,
2348         l.get_id() + '.b.prop': l.pr_opt,
2349         l.get_id() + '.b.qabs': l.qabs,
2350         l.get_id() + '.b.qlost': l.qlost,
2351         l.get_id() + '.b qlbk': l.qlost_brackets,
2352         l.get_id() + '.b.wpwr': l.wasted_power}
2353
2354         self.store_values(row, values)
2355
2356
2357     def show_report(self, keys= None):
2358
2359         self.report_df[keys].plot(
2360             figsize=(20,10), linewidth=5, fontsize=20)
2361         plt.xlabel('Date', fontsize=20)
2362         pd.set_option('display.max_rows', None)
2363         pd.set_option('display.max_columns', None)
2364         pd.set_option('display.width', None)
2365
2366     def save_results(self):
2367
2368         keys = ['DNI', 'elevation', 'zenith', 'azimuth', 'aoi', 'IAM',
2369                 'pr_shadows', 'pr_borders', 'pr_opt_peak', 'solar_fraction']
2370
2371         keys_graphics_power = ['DNI']
2372         keys_graphics_temp = ['DNI']
2373
2374         keys_a = ['NetPower', 'AuxPower', 'GrossPower',
2375                   'SF.a.mf', 'SF.a.tin', 'SF.a.tout',
2376                   'SF.a.pwr', 'SF.a.prth', 'SF.a.globalpr']
2377         keys_graphics_power_a = ['NetPower', 'AuxPower', 'GrossPower',
2378                               'SF.a.pwr']
2379         keys_graphics_temp_a = ['SF.a.mf', 'SF.a.tin', 'SF.a.tout']
2380
2381         keys_x = ['SF.x.mf', 'SF.x.tin', 'SF.x.tout', 'SF.x.pwr',
2382                   'SF.x.prth', 'SF.x.globalpr']
2383         keys_graphics_power_x = ['SF.x.pwr']
2384         keys_graphics_temp_x = ['SF.x.mf', 'SF.x.tout']
2385
2386         keys_b = ['SF.b.tout', 'SF.b.pwr', 'SF.b.wpwr',
2387                   'SF.b.prth', 'SF.b.globalpr']
2388         keys_graphics_power_b = ['SF.b.pwr']
2389         keys_graphics_temp_b = ['SF.b.tout']
2390
2391         if self.datatype == 2:
2392             keys += keys_a
2393             keys_graphics_power += keys_graphics_power_a
2394             keys_graphics_temp += keys_graphics_temp_a
2395
2396         if self.simulation == True:
2397             keys += keys_x
2398             keys_graphics_power += keys_graphics_power_x

```

```

2399     keys_graphics_temp += keys_graphics_temp_x
2400
2401     if self.benchmark == True:
2402         keys += keys_b
2403         keys_graphics_power += keys_graphics_power_b
2404         keys_graphics_temp += keys_graphics_temp_b
2405
2406     self.report_df = self.datasource.dataframe
2407
2408     try:
2409         initialdir = "./simulations_outputs/"
2410         prefix = datetime.today().strftime("%Y%m%d %H%M%S ")
2411         filename_complete = str(self.ID) + "_COMPLETE"
2412         filename_report = str(self.ID) + "_REPORT"
2413         sufix = ".csv"
2414
2415         path_complete = initialdir + prefix + filename_complete + sufix
2416         path_report = initialdir + prefix + filename_report + sufix
2417
2418         self.datasource.dataframe.to_csv(
2419             path_complete, sep=';', decimal = ',')
2420         # self.report_df.to_csv(path_report, sep=';', decimal = ',')
2421
2422     except Exception:
2423         raise
2424         print('Error saving results, unable to save file: %r', path)
2425
2426 def show_message(self):
2427
2428     print("Running simulation for source data file: {0} from: \
2429           {1} to {2}".format(
2430         self.parameters['simulation']['filename'],
2431         self.parameters['simulation']['first_date'],
2432         self.parameters['simulation']['last_date']))
2433     print("Model: {0}".format(
2434         self.parameters['model']['name']))
2435     print("Simulation: {0} ; Benchmark: {1} ; FastMode: {2}".format(
2436         self.parameters['simulation']['simulation'],
2437         self.parameters['simulation']['benchmark'],
2438         self.parameters['simulation']['fastmode']))
2439
2440     print("Site: {0} @ Lat: {1:.2f}º, Long: {2:.2f}º, Alt: {3} m".format(
2441         self.site.name, self.site.latitude,
2442         self.site.longitude, self.site.altitude))
2443
2444     print("Loops:", self.solarfield.total_loops,
2445           'SCA/loop:', self.parameters['loop']['scas'],
2446           'HCE/SCA:', self.parameters['loop']['hces'])
2447     print("SCA model:", self.parameters['SCA']['Name'])
2448     print("HCE model:", self.parameters['HCE']['Name'])
2449     if self.parameters['HTF']['source'] == 'table':
2450         print("HTF form table:", self.parameters['HTF']['name'])
2451     elif self.parameters['HTF']['source'] == 'CoolProp':
2452         print("HTF form CoolProp:", self.parameters['HTF']['CoolPropID'])
2453     print("-----")
2454
2455
2456 class LoopSimulation(object):
2457     """
2458     Definimos la clase simulacion para representar las diferentes

```

```

2459 pruebas que lancemos, variando el archivo TMY, la configuracion del
2460 site, la planta, el modo de operacion o el modelo empleado.
2461 """
2462
2463 def __init__(self, settings):
2464
2465     self.tracking = True
2466     self.htf = None
2467     self.site = None
2468     self.datasource = None
2469     self.parameters = settings
2470
2471     if settings['model']['name'] == 'Barbero4thOrder':
2472         self.model = ModelBarbero4thOrder(settings['model'])
2473     elif settings['model']['name'] == 'Barbero1stOrder':
2474         self.model = ModelBarbero1stOrder(settings['model'])
2475     elif settings['model']['name'] == 'BarberoSimplified':
2476         self.model = ModelBarberoSimplified(settings['model'])
2477
2478     self.datasource = TableData(settings['simulation'])
2479
2480     self.site = Site(settings['site'])
2481
2482     if settings['HTF']['source'] == "CoolProp":
2483         if settings['HTF']['CoolPropID'] not in Fluid._COOLPROP_FLUIDS:
2484             print("Not CoolPropID valid")
2485             sys.exit()
2486         else:
2487             self.htf = FluidCoolProp(settings['HTF'])
2488     else:
2489         self.htf = FluidTabular(settings['HTF'])
2490
2491     self.base_loop = BaseLoop(settings['loop'],
2492                             settings['SCA'],
2493                             settings['HCE'])
2494
2495 def runSimulation(self):
2496
2497     self.show_message()
2498
2499     flag_0 = datetime.now()
2500
2501     for row in self.datasource.dataframe.iterrows():
2502
2503         solarpos = self.site.get_solarposition(row)
2504
2505         if solarpos['zenith'][0] < 90:
2506             self.tracking = True
2507         else:
2508             self.tracking = False
2509
2510         self.simulate_base_loop(solarpos, row)
2511
2512         str_data = ("{} Ang. Zenith: {} DNI: {} W/m2 " +
2513                     "Qm: {}kg/s Tin: {}K Tout: {}K")
2514
2515         print(str_data.format(row[0], solarpos['zenith'][0],
2516                               row[1]['DNI'], self.base_loop.act_massflow,
2517                               self.base_loop.tin, self.base_loop.tout))
2518

```

```

2519     print(self.datasource.dataframe)
2520
2521     flag_1 = datetime.now()
2522     delta_t = flag_1 - flag_0
2523     print("Total runtime: ", delta_t.total_seconds())
2524
2525     self.save_results()
2526
2527 def simulate_base_loop(self, solarpos, row):
2528
2529     values = {'tin': 573,
2530               'pin': 1900000,
2531               'massflow': 4}
2532     self.base_loop.initialize('values', values)
2533     HCE_var = ''
2534     SCA_var = ''
2535
2536     for c in row[1].keys():
2537         if c in self.base_loop.parameters_sca.keys():
2538             SCA_var = c
2539
2540     for c in row[1].keys():
2541         if c in self.base_loop.parameters_hce.keys():
2542             HCE_var = c
2543
2544     for s in self.base_loop.scas:
2545         if SCA_var != '':
2546             s.parameters[SCA_var] = row[1][SCA_var]
2547             aoi = s.get_aoi(solarpos)
2548             for h in s.hces:
2549                 if HCE_var != '':
2550                     h.parameters[HCE_var] = row[1][HCE_var]
2551                     h.set_pr_opt(solarpos)
2552                     h.set_qabs(aoi, solarpos, row)
2553                     h.set_tin()
2554                     h.set_pin()
2555                     h.tout = h.tin
2556                     self.model.calc_pr(h, self.htf, row)
2557
2558             self.base_loop.tout = self.base_loop.scas[-1].hces[-1].tout
2559             self.base_loop.pout = self.base_loop.scas[-1].hces[-1].pout
2560             self.base_loop.set_loop_values_from_HCEs('actual')
2561             print('pr', self.base_loop.pr_act_massflow ,
2562                  'tout', self.base_loop.tout,
2563                  'massflow', self.base_loop.massflow)
2564
2565         if HCE_var + SCA_var != '':
2566             self.datasource.dataframe.at[row[0], HCE_var + SCA_var] = row[1][HCE_var
+ SCA_var]
2567             self.datasource.dataframe.at[row[0], 'pr'] = self.base_loop.pr_act_massflow
2568             self.datasource.dataframe.at[row[0], 'tout'] = self.base_loop.tout
2569             self.datasource.dataframe.at[row[0], 'pout'] = self.base_loop.pout
2570             self.datasource.dataframe.at[row[0], 'Z'] = solarpos['zenith'][0]
2571             self.datasource.dataframe.at[row[0], 'E'] = solarpos['elevation'][0]
2572             self.datasource.dataframe.at[row[0], 'aoi'] = aoi
2573
2574 def save_results(self):
2575
2576     try:

```

```

2578     initialdir = "./simulations_outputs/"
2579     prefix = datetime.today().strftime("%Y%m%d %H%M%S")
2580     filename = "Loop Simulation"
2581     sufix = ".csv"
2582
2583     path = initialdir + prefix + filename + sufix
2584
2585     self.datasource.dataframe.to_csv(path, sep=';', decimal = ',')
2586
2587 except Exception:
2588     raise
2589     print('Error saving results, unable to save file: %r', path)
2590
2591
2592 def show_message(self):
2593
2594     print("Running simulation for source data file: {}".format(
2595         self.parameters['simulation']['filename']))
2596
2597     print("Site: {} @ Lat: {:.2f}°, Long: {:.2f}°, Alt: {} m".format(
2598         self.site.name, self.site.latitude,
2599         self.site.longitude, self.site.altitude))
2600
2601     print('SCA/loop:', self.parameters['loop']['scas'],
2602           'HCE/SCA:', self.parameters['loop']['hces'])
2603     print("SCA model:", self.parameters['SCA']['Name'])
2604     print("HCE model:", self.parameters['HCE']['Name'])
2605     print("HTF:", self.parameters['HTF']['name'])
2606     print("-----")
2607
2608
2609 class Fluid:
2610
2611     _T_REF = 285.856 # Kelvin, T_REF= 12.706 Celsius Degrees
2612     _COOLPROP_FLUIDS = ['Water', 'INCOMP::TVP1', 'INCOMP::S800']
2613
2614     def test_fluid(self, tmax, tmin, p):
2615
2616         data = []
2617
2618         for tt in range(int(round(tmax)), int(round(tmin)), -5):
2619             data.append({'T': tt,
2620                         'P': p,
2621                         'cp': self.get_specific_heat(tt, p),
2622                         'rho': self.get_density(tt, p),
2623                         'mu': self.get_dynamic_viscosity(tt, p),
2624                         'kt': self.get_thermal_conductivity(tt, p),
2625                         'H': self.get_enthalpy(tt, p),
2626                         'T-H': self.get_temperature(self.get_enthalpy(tt, p), p)})
2627         df = pd.DataFrame(data)
2628         print(round(df, 6))
2629
2630     def get_specific_heat(self, p, t):
2631         pass
2632
2633     def get_density(self, p, t):
2634         pass
2635
2636     def get_thermal_conductivity(self, p, t):
2637         pass

```

```
2638
2639     def get_enthalpy(self, p, t):
2640         pass
2641
2642     def get_temperature(self, h, p):
2643         pass
2644
2645     def get_temperature_by_integration(self, tin, q, mf=None, p=None):
2646         pass
2647
2648     def get_dynamic_viscosity(self, t, p):
2649         pass
2650
2651     def get_Reynolds(self, dri, t, p, massflow):
2652
2653         return (4 * massflow /
2654             (np.pi * dri * self.get_dynamic_viscosity(t,p)))
2655
2656     def get_massflow_from_Reynolds(self, dri, t, p, re):
2657
2658         if t > self.tmax:
2659             t = self.tmax
2660
2661         return re * np.pi * dri * self.get_dynamic_viscosity(t,p) / 4
2662
2663     def get_prandtl(self, t, p):
2664
2665         # Specific heat capacity
2666         cp = self.get_specific_heat(t, p)
2667
2668         # Fluid dynamic viscosity
2669         mu = self.get_dynamic_viscosity(t, p)
2670
2671         # Fluid density
2672         rho = self.get_density(t, p)
2673
2674         # Fluid thermal conductivity
2675         kf = self.get_thermal_conductivity(t, p)
2676
2677         # Fluid thermal diffusivity
2678         alpha = kf / (rho * cp)
2679
2680         # # Prandtl number
2681         prandtl = cp * mu / kf
2682
2683         return prandtl
2684
2685 class FluidCoolProp(Fluid):
2686
2687     def __init__(self, settings = None):
2688
2689         if settings['source'] == 'table':
2690             self.name = settings['name']
2691             self.tmax = settings['tmax']
2692             self.tmin = settings['tmin']
2693
2694         elif settings['source'] == 'CoolProp':
2695             self.tmax = PropsSI('T_MAX', settings['CoolPropID'])
2696             self.tmin = PropsSI('T_MIN', settings['CoolPropID'])
2697             self.coolpropID = settings['CoolPropID']
```

```
2698
2699     def get_density(self, t, p):
2700
2701         if t > self.tmax:
2702             t = self.tmax
2703
2704         return PropsSI('D','T',t,'P', p, self.coolpropID)
2705
2706     def get_dynamic_viscosity(self, t, p):
2707
2708         if t > self.tmax:
2709             t = self.tmax
2710
2711         #p = 1600000
2712         return PropsSI('V','T',t,'P', p, self.coolpropID)
2713
2714     def get_specific_heat(self, t, p):
2715
2716         if t > self.tmax:
2717             t = self.tmax
2718
2719         return PropsSI('C','T',t,'P', p, self.coolpropID)
2720
2721     def get_thermal_conductivity(self, t, p):
2722         ''' Saturated Fluid conductivity at temperature t '''
2723
2724         if t > self.tmax:
2725             t = self.tmax
2726
2727         return PropsSI('L','T',t,'P', p, self.coolpropID)
2728
2729     def get_enthalpy(self, t, p):
2730
2731         if t > self.tmax:
2732             t = self.tmax
2733
2734         CP.set_reference_state(self.coolpropID, 'ASHRAE')
2735         deltaH = PropsSI('H','T',t , 'P', p, self.coolpropID)
2736         CP.set_reference_state(self.coolpropID, 'DEF')
2737
2738         return deltaH
2739
2740     def get_delta_enthalpy(self, t1, t2, p1, p2):
2741
2742         CP.set_reference_state(self.coolpropID, 'ASHRAE')
2743         h1 = PropsSI('H','T',t1 , 'P', p1, self.coolpropID)
2744         h2 = PropsSI('H','T',t2 , 'P', p2, self.coolpropID)
2745         CP.set_reference_state(self.coolpropID, 'DEF')
2746
2747         return mf * (h2-h1)
2748
2749     def get_temperature(self, h, p):
2750
2751         CP.set_reference_state(self.coolpropID, 'ASHRAE')
2752         temperature = PropsSI('T', 'H', h, 'P', p, self.coolpropID)
2753         CP.set_reference_state(self.coolpropID, 'DEF')
2754
2755         return temperature
2756
2757     def get_temperature_by_integration(self, t, q, mf = None, p = None):
```

```
2758
2759     if t > self.tmax:
2760         t = self.tmax
2761
2762     CP.set_reference_state(self.coolpropID, 'ASHRAE')
2763     hin = PropsSI('H', 'T', t, 'P', p, self.coolpropID)
2764     try:
2765         temperature = PropsSI('T', 'H', hin + q/mf, 'P', p, self.coolpropID)
2766     except:
2767         temperature = self.tmax
2768     CP.set_reference_state(self.coolpropID, 'DEF')
2769
2770     return temperature
2771
2772 class FluidTabular(Fluid):
2773
2774     def __init__(self, settings=None):
2775
2776         self.name = settings['name']
2777         self.cp = settings['cp']
2778         self.rho = settings['rho']
2779         self.mu = settings['mu']
2780         self.kt = settings['kt']
2781         self.h = settings['h']
2782         self.t = settings['t']
2783         self.tmax = settings['tmax']
2784         self.tmin = settings['tmin']
2785
2786
2787     def get_density(self, t, p):
2788
2789         # Dowtherm A.pdf, 2.2 Single Phase Liquid Properties. pg. 8.
2790
2791         poly = np.polynomial.polynomial.Polynomial(self.rho)
2792
2793         return poly(t)
2794
2795     def get_dynamic_viscosity(self, t, p):
2796
2797         poly = np.polynomial.polynomial.Polynomial(self.mu)
2798
2799         mu = poly(t)
2800
2801         return mu
2802
2803     def get_specific_heat(self, t, p):
2804
2805         poly = np.polynomial.polynomial.Polynomial(self.cp)
2806
2807         return poly(t)
2808
2809     def get_thermal_conductivity(self, t, p):
2810         ''' Saturated Fluid conductivity at temperature t '''
2811
2812         poly = np.polynomial.polynomial.Polynomial(self.kt)
2813
2814         return poly(t)
2815
2816     def get_enthalpy(self, t, p):
```

```

2818     poly = np.polynomial.polynomial.Polynomial(self.h)
2819
2820     return poly(t)
2821
2822 def get_delta_enthalpy(self, t1, t2, p1, p2):
2823
2824     cp0, cp1, cp2, cp3, cp4, cp5, cp6, cp7, cp8 = tuple(self.cp)
2825
2826     h = (
2827         (cp0 * t2 + cp1 * t2**2 / 2 + cp2 * t2**3 / 3 +
2828          cp3 * t2**4 / 4 + cp4 * t2**5 / 5 + cp5 * t2**6 / 6 +
2829          cp6 * t2**7 / 7 + cp7 * t2**8 / 8 + cp8 * t2**9 / 9)
2830         -
2831         (cp0 * t1 + cp1 * t1**2 / 2 + cp2 * t1**3 / 3 +
2832          cp3 * t1**4 / 4 + cp4 * t1**5 / 5 + cp5 * t1**6 / 6 +
2833          cp6 * t1**7 / 7 + cp7 * t1**8 / 8 + cp8 * t1**9 / 9))
2834
2835     return h
2836
2837 def get_temperature(self, h, p):
2838
2839     poly = np.polynomial.polynomial.Polynomial(self.t)
2840
2841     return poly(h)
2842
2843 def get_temperature_by_integration(self, tin, h, mf=None, p=None):
2844
2845
2846     cp0, cp1, cp2, cp3, cp4, cp5, cp6, cp7, cp8 = tuple(self.cp)
2847
2848     a0 = (h/mf + cp0 * tin + cp1 * tin**2 / 2 + cp2 * tin**3 / 3 +
2849           cp3 * tin**4 / 4 + cp4 * tin**5 / 5 + cp5 * tin**6 / 6 +
2850           cp6 * tin**7 / 7 + cp7 * tin**8 / 8 + cp8 * tin**9 / 9)
2851
2852     factors = [a0, -cp0, -cp1 / 2, -cp2 / 3, -cp3 / 4, -cp4 / 5, -cp5 / 6,
2853                -cp6 / 7, -cp7 / 8, -cp8 / 9]
2854
2855     poly = np.polynomial.polynomial.Polynomial(factors)
2856     roots = poly.roots()
2857
2858     tout_bigger = []
2859     tout_smaller = []
2860
2861     for r in roots:
2862         if r.imag == 0.0:
2863             if r.real >= tin:
2864                 tout_bigger.append(r.real)
2865             else:
2866                 tout_smaller.append(r.real)
2867     if h > 0:
2868         tout = min(tout_bigger)
2869     elif h<0:
2870         tout = max(tout_smaller)
2871     else:
2872         tout = tin
2873
2874     return tout
2875
2876
2877 class Weather(object):

```

```
2878
2879     def __init__(self, settings = None):
2880
2881         self.dataframe = None
2882         self.site = None
2883         self.weatherdata = None
2884
2885     if settings is not None:
2886         self.openWeatherDataFile(settings['filepath'] +
2887                                 settings['filename'])
2888         # self.file
2889     else:
2890         self.openWeatherDataFile()
2891
2892     self.dataframe = self.weatherdata[0]
2893     self.site = self.weatherdata[1]
2894
2895     self.change_units()
2896     self.filter_columns()
2897
2898
2899     def openWeatherDataFile(self, path = None):
2900
2901         try:
2902             if path is None:
2903                 root = Tk()
2904                 root.withdraw()
2905                 path = askopenfilename(initialdir = ".weather_files/",
2906                                         title = "choose your file",
2907                                         filetypes = (("TMY files","*.tm2"),
2908                                                     ("TMY files","*.tm3"),
2909                                                     ("csv files","*.csv"),
2910                                                     ("all files","*.*")))
2911                 root.update()
2912                 root.destroy()
2913
2914             if path is None:
2915                 return
2916             else:
2917                 strfilename, strext = os.path.splitext(path)
2918
2919             if strext == ".csv":
2920                 self.weatherdata = pvlib.iotools.tmy.read_tmy3(path)
2921                 self.file = path
2922             elif (strext == ".tm2" or strext == ".tmy"):
2923                 self.weatherdata = pvlib.iotools.tmy.read_tmy2(path)
2924                 self.file = path
2925             elif strext == ".xls":
2926                 pass
2927             else:
2928                 print("unknow extension ", strext)
2929                 return
2930
2931         except Exception:
2932             raise
2933             txMessageBox.showerror('Error loading Weather Data File',
2934                                     'Unable to open file: %r', self.file)
2935
2936     def change_units(self):
2937
```

```

2938     for c in self.dataframe.columns:
2939         if (c == 'DryBulb') or (c == 'DewPoint'): # From Celsius Degrees to K
2940             self.dataframe[c] *= 0.1
2941             self.dataframe[c] += 273.15
2942         if c=='Pressure': # from mbar to Pa
2943             self.dataframe[c] *= 1e2
2944
2945     def filter_columns(self):
2946
2947         needed_columns = ['DNI', 'DryBulb', 'DewPoint', 'Wspd', 'Pressure']
2948         columns_to_drop = []
2949         for c in self.dataframe.columns:
2950             if c not in needed_columns:
2951                 columns_to_drop.append(c)
2952         self.dataframe.drop(columns = columns_to_drop, inplace = True)
2953
2954     def site_to_dict(self):
2955         """
2956             pvlib.iotools realiza modificaciones en los nombres de las columnas.
2957         """
2958
2959         return {"name": 'nombre_site',
2960                 "latitude": self.site['latitude'],
2961                 "longitude": self.site['longitude'],
2962                 "altitude": self.site['altitude']}
2963
2964 class FieldData(object):
2965
2966     def __init__(self, settings, tags = None):
2967         self.filename = settings['filename']
2968         self.filepath = settings['filepath']
2969         self.file = self.filepath + self.filename
2970         self.first_date = pd.to_datetime(settings['first_date'])
2971         self.last_date = pd.to_datetime(settings['last_date'])
2972         self.tags = tags
2973         self.dataframe = None
2974
2975         self.openFieldDataFile(self.file)
2976         self.rename_columns()
2977         self.change_units()
2978
2979
2980     def openFieldDataFile(self, path = None):
2981
2982         ...
2983         fielddata
2984         ...
2985
2986         rows_list = []
2987         index_count = 1 # Skip fist row in skiprows: it has got columns names
2988         #dateparse = lambda x: pd.datetime.strptime(x, '%YYYY/%m/%d %H:%M')
2989         try:
2990             if path is None:
2991                 root = Tk()
2992                 root.withdraw()
2993                 path = askopenfilename(initialdir = ".fielddata_files/",
2994                                         title = "choose your file",
2995                                         filetypes = ((("csv files","*.csv"),
2996                                         ("all files","*.*"))))
2997             root.update()

```

```

2998         root.destroy()
2999     if path is None:
3000         return
3001     else:
3002         strfilename, strext = os.path.splitext(path)
3003         if strext == ".csv":
3004             df = pd.read_csv(
3005                 path, sep=';',
3006                 decimal=',',
3007                 usecols=[0])
3008
3009         df = pd.to_datetime(
3010             df['date'], format = "%d/%m/%Y %H:%M")
3011
3012         for row in df:
3013             if (row < self.first_date or
3014                 row > self.last_date):
3015                 rows_list.append(index_count)
3016             index_count += 1
3017
3018         self.dataframe = pd.read_csv(
3019             path, sep=';',
3020             decimal=',',
3021             dayfirst=True,
3022             index_col=0,
3023             skiprows=rows_list)
3024
3025         self.file = path
3026
3027     else:
3028         print("unknow extension ", strext)
3029         return
3030
3031 except Exception:
3032     raise
3033     txMessageBox.showerror('Error loading FieldData File',
3034                           'Unable to open file: %r', self.file)
3035
3036     self.dataframe.index = pd.to_datetime(self.dataframe.index,
3037                                         format= "%d/%m/%Y %H:%M")
3038
3039 def change_units(self):
3040
3041     for c in self.dataframe.columns:
3042         if ('.a.t' in c) or ('DryBulb' in c) or ('Dew' in c):
3043             self.dataframe[c] += 273.15 # From Celsius Degrees to K
3044         if '.a.p' in c:
3045             self.dataframe[c] *= 1e5 # From Bar to Pa
3046         if 'Pressure' in c:
3047             self.dataframe[c] *= 1e2 # From mBar to Pa
3048
3049 def rename_columns(self):
3050
3051     # Replace tags with names as indicated in configuration file
3052     # (field_data_file: tags)
3053
3054     rename_dict = dict(zip(self.tags.values(), self.tags.keys()))
3055     self.dataframe.rename(columns = rename_dict, inplace = True)
3056
3057

```

```

3058     # Remove unnecessary columns
3059     columns_to_drop = []
3060     for c in self.dataframe.columns:
3061         if c not in self.tags.keys():
3062             columns_to_drop.append(c)
3063     self.dataframe.drop(columns = columns_to_drop, inplace = True)
3064
3065 class TableData(object):
3066
3067     def __init__(self, settings):
3068         self.filename = settings['filename']
3069         self.filepath = settings['filepath']
3070         self.file = self.filepath + self.filename
3071         self.dataframe = None
3072
3073         self.openDataFile(self.file)
3074
3075
3076     def openDataFile(self, path = None):
3077
3078         '''
3079         Table ddata
3080         '''
3081
3082         try:
3083             if path is None:
3084                 root = Tk()
3085                 root.withdraw()
3086                 path = askopenfilename(initialdir = ".data_files/",
3087                                         title = "choose your file",
3088                                         filetypes = (("csv files","*.csv"),
3089                                         ("all files","*.*")))
3090                 root.update()
3091                 root.destroy()
3092             else:
3093                 strfilename, strext = os.path.splitext(path)
3094
3095             if strext == ".csv":
3096                 self.dataframe = pd.read_csv(
3097                     path, sep=';',
3098                     decimal= ',',
3099                     dayfirst=True,
3100                     index_col=0)
3101
3102                 self.file = path
3103             else:
3104                 print("unknow extension ", strext)
3105             return
3106
3107             self.dataframe.index = pd.to_datetime(self.dataframe.index,
3108                                         format= "%d/%m/%Y %H:%M")
3109         except Exception:
3110             raise
3111             txMessageBox.showerror('Error loading FieldData File',
3112                                         'Unable to open file: %r', self.file)
3113
3114
3115 class Site(object):
3116     def __init__(self, settings):
3117

```

```
3118     self.name = settings['name']
3119     self.latitude = settings['latitude']
3120     self.longitude = settings['longitude']
3121     self.altitude = settings['altitude']
3122
3123
3124     def get_solarposition(self, row):
3125
3126         solarpos = pvlib.solarposition.get_solarposition(
3127             row[0] + timedelta(hours=0.5),
3128             self.latitude,
3129             self.longitude,
3130             self.altitude,
3131             pressure=row[1]['Pressure'],
3132             temperature=row[1]['DryBulb'])
3133
3134         return solarpos
3135
3136     def get_hour_angle(self, row, equation_of_time):
3137
3138         hour_angle = pvlib.solarposition.hour_angle(
3139             row[0] + timedelta(hours=0.5),
3140             self.longitude,
3141             equation_of_time)
3142
3143         return hour_angle
3144
3145 if __name__ == '__main__':
3146
3147     with open("./saved_configurations/TEST_2016_DOWA.json") as simulation_file:
3148         SIMULATION_SETTINGS = json.load(simulation_file)
3149         SIM = cs.SolarFieldSimulation(SIMULATION_SETTINGS)
3150
3151         FLAG_00 = datetime.now()
3152         SIM.runSimulation()
3153         FLAG_01 = datetime.now()
3154         DELTA_01 = FLAG_01 - FLAG_00
3155         print("Total runtime: ", DELTA_01.total_seconds())
```

ANEXO C. CÓDIGO FUENTE: INTERFACE.PY

```

1 # -*- coding: utf-8 -*-
2
3 """
4 interface.py: A Tkinter application for creating configuration files to
5 run simulation with csenergy.py
6 @author: pacomunuera
7 2020
8 """
9
10
11 import sys
12 sys.path.append('./libs')
13 import os.path
14 import csenergy as cs
15 import CoolProp.CoolProp as CP
16 import pvlib as pvlib
17 import tkinter as tk
18 from tkinter.filedialog import askopenfilename, asksaveasfile
19 import tkinter.ttk as ttk
20 from tkinter import messagebox
21 # recipe-580746-1.py from
22 # http://code.activestate.com/recipes/
23 # 580746-t kinter-treeview-like-a-table-or-multicolumn-listb/
24 import recipe5807461 as table
25 import json
26 from decimal import Decimal
27 from datetime import datetime
28 import pandas as pd
29
30
31 class Interface(object):
32
33     _COOLPROP_FLUIDS = ['Water', 'INCOMP::TVP1', 'INCOMP::S800']
34
35     _MODELS = ['Barbero4thOrder', 'Barbero1stOrder', 'BarberoSimplified']
36
37     _DIR = {
38         'saved_configurations': './saved_configurations/',
39         'site_files': './site_files/',
40         'fluid_files': './fluid_files',
41         'hce_files': './hce_files',
42         'sca_files': './sca_files',
43         'fielddata_files': './fielddata_files',
44         'weather_files': './weather_files'}
45
46     cfg_settings = {
47         'simulation': {},
48         'solar_plant': {},
49         'site': {},
50         'weather': {},
51         'hce': {},
52         'SCA': {},
53         'hot_hce': {},
54         'cold_hce': {},
55         'cycle': {},
56         'hce_model_settings': {},
57         'hce_scattered_params': {},
58         'sca_scattered_params': {}}
59
60     def __init__(self):

```

```

61
62     # Main window
63     self.root = tk.Tk()
64     self.root.attributes('-fullscreen', False)
65     self.varroottitle = tk.StringVar(self.root)
66     self.root.title('Solar Field Configurator ')
67     w, h = self.root.winfo_screenwidth(), self.root.winfo_screenheight()
68     self.root.geometry('%dx%d+0+0' % (w-200, h-200))
69     self.root.geometry('%dx%d+0+0' % (800, 800))
70
71     # Menu
72     self.menubar = tk.Menu(self.root)
73
74     self.simulation_menu = tk.Menu(self.menubar, tearoff=0)
75     self.simulation_menu.add_command(
76         label='New', command=self.simulation_new)
77     self.simulation_menu.add_command(
78         label='Open', command=self.simulation_open)
79     self.simulation_menu.add_command(
80         label='Save', command=self.simulation_save)
81     self.simulation_menu.add_command(
82         label='Save as...', command=self.simulation_save_as)
83     self.simulation_menu.add_separator()
84     self.simulation_menu.add_command(
85         label='Exit', command=self.simulation_exit)
86     self.menubar.add_cascade(
87         label='Simulation', menu=self.simulation_menu)
88     self.run_menu = tk.Menu(self.menubar, tearoff=0)
89     self.run_menu.add_command(label='Run', command=self.run_simulation)
90     self.menubar.add_cascade(label='Run', menu=self.run_menu)
91     self.help_menu = tk.Menu(self.menubar, tearoff=0)
92     self.help_menu.add_command(label='Help', command=self.help_help)
93     self.help_menu.add_command(label='About', command=self.help_about)
94     self.menubar.add_cascade(label='Help', menu=self.help_menu)
95     self.root.config(menu=self.menubar)
96
97     # Notebook (tabs)
98     self.nb = ttk.Notebook(self.root)
99
100    self.fr_simulation = tk.Frame(self.nb)
101    self.fr_solarfield = tk.Frame(self.nb)
102    self.fr_fluid = tk.Frame(self.nb)
103    self.fr_hce = tk.Frame(self.nb)
104    self.fr_sca = tk.Frame(self.nb)
105
106    self.buildNotebook()
107    self.buildSolarFieldFrame()
108    self.buildSimulationFrame()
109    self.buildFluidFrame()
110    self.buildHCEFrame()
111    self.buildSCAFrame()
112
113    self.simulation_open(self._DIR['saved_configurations']+ 'template.json')
114
115    def simulation_new(self):
116
117        # Simulation configuration
118        self.varsimID.set('')
119        self.varsimdatatype.set(1)
120        self.varsimulation.set(False)

```

```

121     self.varbenchmark.set(False)
122     self.varfastmode.set(False)
123     self.vardatafilename.set('')
124     self.vardatafilepath.set('')
125     self.vardatafileurl.set('')
126     self.varfirstdate.set(pd.to_datetime('2000/01/01 1:00').strftime(
127         '%Y/%m/%d %H:%M'))
128     self.varlastdate.set(pd.to_datetime('2000/12/31 1:00').strftime(
129         '%Y/%m/%d %H:%M'))
130     self.cmbmodelname.set('')
131     self.varmodelmaxerrt.set(1.0)
132     self.varmodelmaxerrtro.set(0.1)
133     self.varmodelmaxerrpr.set(0.01)
134
135     self.varsitename.set(0)
136     self.varsitelat.set(0)
137     self.varsitelong.set(0)
138     self.varsitealt.set(0)
139     self.checkoptions()
140     self.checkfastmode()
141
142     # Solarfield configuration
143     self.solarfield_table.table_data = []
144     self.columns_table.table_data = []
145
146     self.vartin.set(0)
147     self.vartout.set(0)
148     self.varpin.set(0)
149     self.varpout.set(0)
150     self.vartmin.set(0)
151     self.vartmax.set(0)
152     self.varratedmassflow.set(0)
153     self.varrecirculation.set(0)
154     self.varscas.set(0)
155     self.varhces.set(0)
156     self.varrowspacing.set(0)
157     self.varscatrackingtype.set(1)
158     self.varfluidtable.set(1)
159     self.varfluidname.set('')
160     self.varfluidtmax.set(0)
161     self.varfluidtmin.set(0)
162     self.fluid_table.table_data = []
163     self.cmbcoolpropID.set('')
164     self.checkfluid()
165
166     # SCA configuration
167     self.varscaname.set('')
168     self.varscalength.set(0)
169     self.varscaperture.set(0)
170     self.varscafocallen.set(0)
171     self.varscIAMF0.set(0)
172     self.varscIAMF1.set(0)
173     self.varscIAMF2.set(0)
174     self.varscareflectance.set(0)
175     self.varscageoaccuracy.set(0)
176     self.varscatracktwist.set(0)
177     self.varscacleanliness.set(0)
178     self.varscafactor.set(0)
179     self.varscavailability.set(0)
180

```

```

181     # HCE configuration
182     self.varhcename.set('')
183     self.varhcedri.set(0)
184     self.varhcedro.set(0)
185     self.varhcedgi.set(0)
186     self.varhcedgo.set(0)
187     self.varhcelength.set(0)
188     self.varhceemittanceA0.set(0)
189     self.varhceemittanceA1.set(0)
190     self.varhceabsorptance.set(0)
191     self.varhcetransmittance.set(0)
192     self.varhceinnerroughness.set(0)
193     self.varhceminreynolds.set(0)
194     self.varhcebrackets.set(0)
195     self.updateHCEperSCA()
196
197     self.fr_fluid.update()
198     self.fr_hce.update()
199     self.fr_solarfield.update()
200     self.fr_sca.update()
201     self.fr_simulation.update()
202
203 def simulation_open(self, path=None):
204
205     if path == None:
206         path = askopenfilename(initialdir=self._DIR['saved_configurations'],
207                               title='choose your file',
208                               filetypes=[('JSON files', '*.json')])
209
210     head, tail = os.path.split(path)
211     self.filename = path
212     self.root.title('Solar Field Configurator: ' + tail)
213     with open(path) as cfg_file:
214         cfg = json.load(cfg_file,
215                         parse_float= float,
216                         parse_int= int)
217
218     # Simulation configuration
219     self.varsimID.set(cfg['simulation']['ID'])
220     self.varsimdatatype.set(cfg['simulation']['datatype'])
221     self.varsimulation.set(cfg['simulation']['simulation'])
222     self.varbenchmark.set(cfg['simulation']['benchmark'])
223     self.varfastmode.set(cfg['simulation']['fastmode'])
224     self.vardatafilename.set(cfg['simulation']['filename'])
225     self.vardatafilepath.set(cfg['simulation']['filepath'])
226     self.vardatafileurl.set(cfg['simulation']['filepath'] +
227                             cfg['simulation']['filename'])
228
229     self.varfirstdate.set(pd.to_datetime(
230         cfg['simulation']['first_date']))
231     self.varlastdate.set(pd.to_datetime(
232         cfg['simulation']['last_date']))
233
234     self.cmbmodelname.set(cfg['model']['name'])
235     self.varmodelmaxerrt.set(cfg['model']['max_err_t'])
236     self.varmodelmaxerrtro.set(cfg['model']['max_err_tro'])
237     self.varmodelmaxerrpr.set(cfg['model']['max_err_pr'])
238
239     self.varsitename.set(cfg['site']['name'])
240     self.varsitelat.set(cfg['site']['latitude'])

```

```

241     self.varsitelong.set(cfg['site']['longitude'])
242     self.varsitealt.set(cfg['site']['altitude'])
243     self.checkoptions()
244     self.checkfastmode()
245
246     # Solarfield configuration
247     list_subfields = []
248     for r in cfg['subfields']:
249         list_subfields.append(list(r.values()))
250
251     self.solarfield_table.table_data = list_subfields
252
253     if 'tags' in cfg.keys():
254         list_tags = []
255         for r in cfg['tags']:
256             list_tags.append([r, ' ', cfg['tags'][r]])
257         self.columns_table.table_data = list_tags
258
259         self.vartin.set(cfg['loop']['rated_tin'])
260         self.vartout.set(cfg['loop']['rated_tout'])
261         self.varpin.set(cfg['loop']['rated_pin'])
262         self.varpout.set(cfg['loop']['rated_pout'])
263         self.vartmin.set(cfg['loop']['tmin'])
264         self.vartmax.set(cfg['loop']['tmax'])
265         self.varratedmassflow.set(cfg['loop']['rated_massflow'])
266         self.varrecirculation.set(cfg['loop']['min_massflow'])
267         self.varscas.set(cfg['loop']['scas'])
268         self.varhces.set(cfg['loop']['hces'])
269         self.varrowspacing.set(cfg['loop']['row_spacing'])
270         self.varscatrackingtype.set(cfg['loop']['Tracking Type'])
271
272     # HTF configuration
273     fluid_table = []
274
275     if cfg['HTF']['source'] == 'table':
276         self.varfluidtable.set(1)
277         self.varfluidname.set(cfg['HTF']['name'])
278         self.varfluidtmax.set(cfg['HTF']['tmax'])
279         self.varfluidtmin.set(cfg['HTF']['tmin'])
280         fluid_table.append(['mu'] + cfg['HTF']['mu'])
281         fluid_table.append(['cp'] + cfg['HTF']['cp'])
282         fluid_table.append(['rho'] + cfg['HTF']['rho'])
283         fluid_table.append(['kt'] + cfg['HTF']['kt'])
284         fluid_table.append(['h'] + cfg['HTF']['h'])
285         fluid_table.append(['t'] + cfg['HTF']['t'])
286         self.fluid_table.table_data = fluid_table
287
288     elif cfg['HTF']['source'] == 'CoolProp':
289         self.varfluidtable.set(2)
290         self.cmbcoolpropID.set(cfg['HTF']['CoolPropID'])
291         self.varfluidtmax.set(cfg['HTF']['tmax'])
292         self.varfluidtmin.set(cfg['HTF']['tmin'])
293
294     self.checkfluid()
295
296     # SCA configuration
297     self.varscaname.set(cfg['SCA']['Name'])
298     self.varscalength.set(cfg['SCA']['SCA Length'])
299     self.varscaperture.set(cfg['SCA']['Aperture'])
300     self.varscafocallen.set(cfg['SCA']['Focal Len'])

```

```

301     self.varscaIAMF0.set(cfg['SCA']['IAM Coefficient F0'])
302     self.varscaIAMF1.set(cfg['SCA']['IAM Coefficient F1'])
303     self.varscaIAMF2.set(cfg['SCA']['IAM Coefficient F2'])
304     self.varscareflectance.set(cfg['SCA']['Reflectance'])
305     self.varscageoaccuracy.set(cfg['SCA']['Geom.Accuracy'])
306     self.varscatracktwist.set(cfg['SCA']['Track Twist'])
307     self.varscacleanliness.set(cfg['SCA']['Cleanliness'])
308     self.varscafactor.set(cfg['SCA']['Factor'])
309     self.varscavailability.set(cfg['SCA']['Availability'])
310
311     # HCE configuration
312     self.varhcename.set(cfg['HCE']['Name'])
313     self.varhcedri.set(cfg['HCE']['Absorber tube inner diameter'])
314     self.varhcedro.set(cfg['HCE']['Absorber tube outer diameter'])
315     self.varhcedgi.set(cfg['HCE']['Glass envelope inner diameter'])
316     self.varhcedgo.set(cfg['HCE']['Glass envelope outer diameter'])
317     self.varhcelength.set(cfg['HCE']['Length'])
318     self.varbellowsratio.set(cfg['HCE']['Bellows ratio'])
319     self.varshieldshading.set(cfg['HCE']['Shield shading'])
320     self.varhceemittanceA0.set(cfg['HCE']['Absorber emittance factor A0'])
321     self.varhceemittanceA1.set(cfg['HCE']['Absorber emittance factor A1'])
322     self.varhceabsorptance.set(cfg['HCE']['Absorber absorptance'])
323     self.varhcetransmittance.set(cfg['HCE']['Envelope transmittance'])
324     self.varhceinnerroughness.set(cfg['HCE']['Inner surface roughness'])
325     self.varhceminreynolds.set(cfg['HCE']['Min Reynolds'])
326     self.varhcebrackets.set(cfg['HCE']['Brackets'])
327     self.updateHCEperSCA()
328
329     self.fr_fluid.update()
330     self.fr_hce.update()
331     self.fr_solarfield.update()
332     self.fr_sca.update()
333     self.fr_simulation.update()
334
335 def simulation_save(self):
336
337     self.save_as_JSON(self.generate_json(), self.filename)
338
339
340 def simulation_save_as(self):
341
342     self.save_as_JSON(self.generate_json())
343
344 def generate_json(self):
345
346     self.tagslist = []
347
348     cfg = dict({'simulation': {},
349                 'model': {},
350                 'site': {},
351                 'loop': {},
352                 'SCA': {},
353                 'HCE': {},
354                 'HTF': {}})
355
356
357     cfg['simulation'][ID] = self.varsimID.get()
358     cfg['simulation'][datatype] = self.varsimdatatype.get()
359     cfg['simulation'][simulation] = self.varsimulation.get()
360     cfg['simulation'][benchmark] = self.varbenchmark.get()

```

```

361     cfg['simulation']['fastmode'] = self.varfastmode.get()
362     cfg['simulation']['filename'] = self.vardatafilename.get()
363     cfg['simulation']['filepath'] = self.vardatafilepath.get()
364     cfg['simulation']['first_date'] = pd.to_datetime(
365         self.varfirstdate.get()).strftime(
366             '%Y/%m/%d %H:%M%z')
367     cfg['simulation']['last_date'] = pd.to_datetime(
368         self.varlastdate.get()).strftime(
369             '%Y/%m/%d %H:%M%z')
370
371     cfg['model']['name'] = self.cmbmodelname.get()
372     cfg['model']['max_err_t'] = self.varmodelmaxerrt.get()
373     cfg['model']['max_err_tro'] = self.varmodelmaxerrtro.get()
374     cfg['model']['max_err_pr'] = self.varmodelmaxerrpr.get()
375
376     # Site configuration
377     cfg['site']['name'] = self.varsitename.get()
378     cfg['site']['latitude'] = self.varsitelat.get()
379     cfg['site']['longitude'] = self.varsitelong.get()
380     cfg['site']['altitude'] = self.varsitealt.get()
381
382     # HTF configuration
383     if self.varfluidtable.get() == 1:
384         cfg['HTF']['source'] = 'table'
385         cfg['HTF']['name'] = self.varfluidname.get()
386
387         parameters_table = list(self.fluid_table.table_data)
388
389         for r in parameters_table:
390             param_name = r[0]
391             param_values = r[1:]
392             param_values = list(map(self.to_number, r[1:]))
393             cfg['HTF'].update(dict({param_name : param_values}))
394
395         cfg['HTF'].update({'tmax' : float(self.entmax.get())})
396         cfg['HTF'].update({'tmin' : float(self.entmin.get())})
397
398     elif self.varfluidtable.get() == 2:
399         cfg['HTF']['source'] = 'CoolProp'
400         cfg['HTF']['CoolPropID'] = self.cmbcoolpropID.get()
401         cfg['HTF']['tmax'] = float(self.varcoolproptmax.get())
402         cfg['HTF']['tmin'] = float(self.varcoolproptmin.get())
403
404
405     # SCA configuration
406     cfg['SCA']['Name'] = self.varscaname.get()
407     cfg['SCA']['SCA Length'] = self.varscalength.get()
408     cfg['SCA']['Aperture'] = self.varscaperture.get()
409     cfg['SCA']['Focal Len'] = self.varscafocallen.get()
410     cfg['SCA']['IAM Coefficient F0'] = self.varsc IAMF0.get()
411     cfg['SCA']['IAM Coefficient F1'] = self.varsc IAMF1.get()
412     cfg['SCA']['IAM Coefficient F2'] = self.varsc IAMF2.get()
413     cfg['SCA']['Track Twist'] = self.varscatracktwist.get()
414     cfg['SCA']['Geom.Accuracy'] = self.varscageoaccuracy.get()
415     cfg['SCA']['Reflectance'] = self.varscareflectance.get()
416     cfg['SCA']['Cleanliness'] = self.varscacleanliness.get()
417     cfg['SCA']['Factor'] = self.varscafactor.get()
418     cfg['SCA']['Availability'] = self.varscavailability.get()
419
420     # HCE configuration

```

```

421     cfg['HCE']['Name'] = self.varhcename.get()
422     cfg['HCE']['Length'] = self.varhcelength.get()
423     cfg['HCE']['Bellows ratio'] = self.varbellowsratio.get()
424     cfg['HCE']['Shield shading'] = self.varshieldshading.get()
425     cfg['HCE']['Absorber tube inner diameter'] = self.varhcedri.get()
426     cfg['HCE']['Absorber tube outer diameter'] = self.varhcedro.get()
427     cfg['HCE']['Glass envelope inner diameter'] = self.varhcedgi.get()
428     cfg['HCE']['Glass envelope outer diameter'] = self.varhcedgo.get()
429     cfg['HCE']['Min Reynolds'] = self.varhceminreynolds.get()
430     cfg['HCE']['Inner surface roughness'] = self.varhceinnerroughness.get()
431     cfg['HCE']['Envelope transmittance'] = self.varhcetransmittance.get()
432     cfg['HCE']['Absorber emittance factor A0'] = self.varhceemittanceA0.get()
433     cfg['HCE']['Absorber emittance factor A1'] = self.varhceemittanceA1.get()
434     cfg['HCE']['Absorber absorptance'] = self.varhceabsorptance.get()
435     cfg['HCE']['Brackets'] = self.varhcebrackets.get()
436
437
438 # loop Configuration
439 cfg['loop']['scas'] = self.varscas.get()
440 cfg['loop']['hces'] = self.varhces.get()
441 cfg['loop']['rated_tin'] = self.vartin.get()
442 cfg['loop']['rated_tout'] = self.vartout.get()
443 cfg['loop']['rated_pin'] = self.varpin.get()
444 cfg['loop']['rated_pout'] = self.varpout.get()
445 cfg['loop']['tmin'] = self.vartmin.get()
446 cfg['loop']['tmax'] = self.vartmax.get()
447 cfg['loop']['rated_massflow'] = self.varratedmassflow.get()
448 cfg['loop']['min_massflow'] = self.varrecirculation.get()
449 cfg['loop']['row_spacing'] = self.varrowspacing.get()
450 cfg['loop']['Tracking Type'] = self.varscatrackingtype.get()
451
452
453 datarow=list(self.solarfield_table.table_data)
454 dictkeys =['name', 'loops']
455
456 subfields = []
457
458 for r in datarow:
459     sf = {}
460     index = 0
461     for v in r:
462         k = dictkeys[index]
463         sf[k]= self.to_number(v)
464         index += 1
465     subfields.append(sf)
466
467 cfg.update({'subfields': subfields})
468
469 if self.varsimdatatype.get() == 2:
470
471     cfg['tags'] = dict({})
472
473     for r in self.columns_table.table_data:
474         cfg['tags'][r[0]] = r[2]
475
476 return cfg
477
478 def save_as_JSON(self, cfg, filename=None):
479
480     if filename is None:

```

```

481         f = asksaveasfile(initialdir=self._DIR['saved_configurations'],
482                             title='choose your file name',
483                             filetypes=[('JSON files', '*.json')],
484                             defaultextension='json')
485     else:
486         f = open(filename, 'w')
487
488     if f is not None:
489         f.write(json.dumps(cfg,
490                            indent= True,
491                            ensure_ascii=False))
492         f.close()
493     else:
494         pass
495
496
497 def __insert_rows__(self, table):
498
499     lst = []
500     for c in table._multicolumn_listbox._columns:
501         lst.append('')
502
503     rows = table.number_of_rows
504     table.insert_row(lst, index = rows)
505
506 def __del_rows__(self, table):
507     table.delete_all_selected_rows()
508
509 def run_simulation(self):
510     # import subprocess
511     import threading
512     try:
513         cfg = self.generate_json()
514
515         SIM = cs.SolarFieldSimulation(cfg)
516         FLAG_00 = datetime.now()
517         hilo = threading.Thread(target=SIM.runSimulation())
518         hilo.start()
519         FLAG_01 = datetime.now()
520         DELTA_01 = FLAG_01 - FLAG_00
521
522     except Exception as e:
523         print("serialization failed", e)
524
525 def help_help():
526     pass
527
528 def help_about():
529     pass
530
531 def simulation_exit(self):
532
533     self.root.destroy()
534
535 def to_number(self, s):
536
537     try:
538         i = int(s)
539         return(i)
540     except ValueError:

```

```

541         pass
542     try:
543         f = float(s)
544         return(f)
545     except ValueError:
546         pass
547     return s
548
549
550 def buildNotebook(self):
551
552     self.nb.add(self.fr_simulation, text='Simulation Configuration', padding=2)
553     self.nb.add(self.fr_solarfield, text=' Solar Field Layout ', padding=2)
554     self.nb.add(self.fr_fluid, text='          HTF          ', padding=2)
555     self.nb.add(self.fr_sca, text='SCA: Solar Collector Assembly', padding=2)
556     self.nb.add(self.fr_hce, text='HCE: Heat Collector Element', padding=2)
557     self.nb.select(self.fr_simulation)
558     self.nb.enable_traversal()
559     self.nb.pack()
560
561 # Simulaton Configuration tab
562 def simulationLoadDialog(self, title, labeltext=''):
563
564     path = askopenfilename(initialdir = self._DIR['saved_configurations'],
565                           title='choose your file',
566                           filetypes=[('JSON files', '*.json')])
567
568     with open(path) as cfg_file:
569         cfg = json.load(cfg_file,
570                         parse_float= float,
571                         parse_int= int)
572
573     self.varsimID.set(cfg['simulation']['ID'])
574     self.varsimdatatype.set(cfg['simulation']['datatype'])
575     self.varsimulation.set(cfg['simulation']['simulation'])
576     self.varbenchmark.set(cfg['simulation']['benchmark'])
577     self.varfastmode.set(cfg['simulation']['fastmode'])
578
579 def checkoptions(self):
580
581     var = self.varsimdatatype.get()
582
583     if var == 1: # Weather File
584         self.varbenchmark.set(False)
585         self.cbbenchmark['state'] ='disabled'
586         self.cbloadsitedata['state'] = 'normal'
587         self.bttagswizard['state']= 'disabled'
588         self.btloadtags['state']= 'disabled'
589         # self.tags_table.state('disabled')
590     elif var == 2: # Field Data File
591         self.varloadsitedata.set(False)
592         self.cbbenchmark['state']= 'normal'
593         self.cbloadsitedata['state'] = 'disabled'
594         self.bttagswizard['state']= 'normal'
595         self.btloadtags['state']= 'normal'
596         # self.tags_table.state(( 'active'))
597     else:
598         pass
599
600 def checkfastmode(self):

```

```
601
602     var = self.varfastmode.get()
603
604     if var:
605         self.varfastmodetext.set('ON')
606     else:
607         self.varfastmodetext.set('OFF')
608
609 def buildSimulationFrame(self):
610
611     self.varsimID = tk.StringVar(self.fr_simulation)
612     self.varsimdatatype = tk.IntVar(self.fr_simulation)
613     self.tagslist = []
614     self.varsimulation = tk.BooleanVar(self.fr_simulation)
615     self.varbenchmark = tk.BooleanVar(self.fr_simulation)
616     self.varfastmode = tk.BooleanVar(self.fr_simulation)
617     self.varfastmodetext = tk.StringVar(self.fr_simulation)
618     self.varfirstdate = tk.StringVar(self.fr_simulation)
619     self.varlastdate = tk.StringVar(self.fr_simulation)
620
621     self.varmodelmaxerrtro = tk.DoubleVar(self.fr_simulation)
622     self.varmodelmaxerrrt = tk.DoubleVar(self.fr_simulation)
623     self.varmodelmaxerrpr = tk.DoubleVar(self.fr_simulation)
624
625     self.varfastmode.set(False)
626     self.checkfastmode()
627
628     self.varloadsitedata = tk.BooleanVar(self.fr_simulation)
629     self.varloadsitedata.set(False)
630
631     self.varsitename = tk.StringVar(self.fr_simulation)
632     self.varsitelat = tk.DoubleVar(self.fr_simulation)
633     self.varsitelong = tk.DoubleVar(self.fr_simulation)
634     self.varsitealt = tk.DoubleVar(self.fr_simulation)
635
636     self.vardatafileurl = tk.StringVar(self.fr_simulation)
637     self.vardatafilename = tk.StringVar(self.fr_simulation)
638     self.vardatafilepath = tk.StringVar(self.fr_simulation)
639
640     self.lbsimID = ttk.Label(
641         self.fr_simulation,
642         text='Simulation ID').grid(
643             row=0, column=0, sticky='W', padx=2, pady=5)
644     self.ensimID = ttk.Entry(
645         self.fr_simulation,
646         textvariable=self.varsimID).grid(
647             row=0, column=1, sticky='W', padx=2, pady=5)
648
649     self.lbmodelname = ttk.Label(
650         self.fr_simulation,
651         text='Model name').grid(
652             row=0, column=2, sticky='E', padx=2, pady=5)
653
654     self.cmbmodelname = ttk.Combobox(self.fr_simulation)
655     self.cmbmodelname['values'] = self._MODELS
656     self.cmbmodelname['state'] = 'readonly'
657     self.cmbmodelname.current(0)
658     self.cmbmodelname.grid(row=0, column=3, sticky='W', padx=2, pady=5)
659
660     self.frame2= ttk.Frame(self.fr_simulation)
```

```

661     self.frame2.grid(row=1, column=0, columnspan=6, padx=2, pady=5)
662
663     self.lbmodelmaxerrtro = ttk.Label(
664         self.frame2,
665         text='Max. Err Tro').grid(
666             row=1, column=0, sticky='W', padx=2, pady=5)
667
668     self.enmodelmaxerrtro = ttk.Entry(
669         self.frame2, textvariable=self.varmodelmaxerrtro).grid(
670             row=1, column=1, sticky='W', padx=2, pady=5)
671
672     self.lbmodelmaxerrrt = ttk.Label(
673         self.frame2,
674         text='Max. Err Tout').grid(
675             row=1, column=2, sticky='W', padx=2, pady=5)
676
677     self.enmodelmaxerrrt = ttk.Entry(
678         self.frame2, textvariable=self.varmodelmaxerrrt).grid(
679             row=1, column=3, sticky='W', padx=2, pady=5)
680
681     self.lbmodelmaxerrpr = ttk.Label(
682         self.frame2,
683         text='Max. Err PR').grid(
684             row=1, column=4, sticky='W', padx=2, pady=5)
685
686     self.enmodelmaxerrpr = ttk.Entry(
687         self.frame2, textvariable=self.varmodelmaxerrpr).grid(
688             row=1, column=5, sticky='W', padx=2, pady=5)
689
690     self.lbdatatype = ttk.Label(
691         self.fr_simulation,
692         text='Choose a Data Source Type:').grid(
693             row=2, column=0, columnspan = 2, sticky='W', padx=2, pady=5)
694
695     self.rbweather = tk.Radiobutton(
696         self.fr_simulation,
697         padx=5,
698         text = 'Weather File',
699         variable=self.varsimdatatype, value=1,
700         command=lambda: self.checkoptions()).grid(
701             row=3, column=0, sticky='W', padx=2, pady=5)
702
703 # RadioButton for Field Data File
704     self.rbfielddata = tk.Radiobutton(
705         self.fr_simulation,
706         padx=5,
707         text = 'Field Data File',
708         variable=self.varsimdatatype, value=2,
709         command=lambda: self.checkoptions()).grid(
710             row=3, column=1, sticky='W', padx=2, pady=5)
711
712     self.btselectdatasource = ttk.Button(
713         self.fr_simulation, text='Select File',
714         command=lambda: self.dataLoadDialog(
715             'Select File',
716             labeltext = 'Select File'))
717     self.btselectdatasource.grid(
718             row=4, column=0, sticky='W', padx=2, pady=5)
719
720 # Data source path

```

```
721     self.vardatafileurl.set('Data source file path...')  
722     self.lbdatasourcepath = ttk.Label(  
723         self.fr_simulation, textvariable=self.vardatafileurl).grid(  
724             row=4, column= 1, columnspan=4, sticky='W', padx=2, pady=5)  
725  
726     self.lbfirstdate = ttk.Label(  
727         self.fr_simulation, text='First Date').grid(  
728             row=5, column=0,sticky='W', padx=2, pady=5)  
729     self.enfirstdate = ttk.Entry(  
730         self.fr_simulation, textvariable=self.varfirstdate).grid(  
731             row=5, column=1, sticky='W', padx=2, pady=5)  
732  
733     self.lblastdate = ttk.Label(  
734         self.fr_simulation, text='Last Date').grid(  
735             row=5, column=2,sticky='E', padx=2, pady=5)  
736     self.enlastdate = ttk.Entry(  
737         self.fr_simulation, textvariable=self.varlastdate).grid(  
738             row=5, column=3, sticky='W', padx=2, pady=5)  
739  
740 # Checkbox for Simulation  
741     self.lbsimulation = ttk.Label(  
742         self.fr_simulation, text='Run test type...').grid(  
743             row=6, column=0, sticky='W', padx=2, pady=5)  
744     self.cbsimulation = ttk.Checkbutton(  
745         self.fr_simulation,  
746             text='Simulation',  
747             variable=self.varsimulation)  
748     self.cbsimulation.grid(  
749             row=6, column=1, sticky='W', padx=2, pady=5)  
750  
751     self.cbbenchmark = ttk.Checkbutton(  
752         self.fr_simulation,  
753             text='Benchmark',  
754             variable=self.varbenchmark)  
755     self.cbbenchmark.grid(  
756             row=6, column=2, sticky='W', padx=2, pady=5)  
757  
758     self.lbfastmode = ttk.Label(  
759         self.fr_simulation, text='Fast mode').grid(  
760             row=7, column=0, sticky='W', padx=2, pady=5)  
761     self.cbfastmode = ttk.Checkbutton(  
762         self.fr_simulation,  
763             textvariable=self.varfastmodetext,  
764             variable= self.varfastmode,  
765             command=lambda: self.checkfastmode()).grid(  
766             row=7, column=1, sticky='W', padx=2, pady=5)  
767  
768     self.lbsitename = ttk.Label(  
769         self.fr_simulation, text='Site').grid(  
770             row=8, column=0, sticky='W', padx=2, pady=5)  
771     self.ensitename = ttk.Entry(  
772         self.fr_simulation, textvariable=self.varsitename).grid(  
773             row=8, column=1, sticky='W', padx=2, pady=5)  
774  
775     self.cbloadsitedata = ttk.Checkbutton(  
776         self.fr_simulation,  
777             text='Load site data from weather file',  
778             variable=self.varloadsitedata)  
779     self.cbloadsitedata.grid(  
780             row=8, column=2, sticky='W', padx=2, pady=5)
```

```

781
782     self.lbsitelat = ttk.Label(
783         self.fr_simulation, text='Latitude [°]').grid(
784             row=10, column=0, sticky='W', padx=2, pady=5)
785     self.ensitelat = ttk.Entry(
786         self.fr_simulation, textvariable=self.varsitelat).grid(
787             row=10, column=1, sticky='W', padx=2, pady=5)
788     self.lbsitelong = ttk.Label(
789         self.fr_simulation, text='Longitude [°']).grid(
790             row=11, column=0, sticky='W', padx=2, pady=5)
791     self.ensitelong = ttk.Entry(
792         self.fr_simulation, textvariable=self.varsitelong).grid(
793             row=11, column=1, sticky='W', padx=2, pady=5)
794     self.lbsitealt = ttk.Label(
795         self.fr_simulation, text='Altitude [m]').grid(
796             row=12, column=0, sticky='W', padx=2, pady=5)
797     self.ensitealt = ttk.Entry(
798         self.fr_simulation, textvariable=self.varsitealt).grid(
799             row=12, column=1, sticky='W', padx=2, pady=5)
800
801     self.varsimdatatype.set(1) # 1 for Weather File, 2 for Field Data File
802     self.checkoptions()
803     self.fr_solarfield.update()
804
805
806 def solarfield_save_dialog(self, title, labeltext = '') :
807
808     #encoder.FLOAT_REPR = lambda o: format(o, '.2f')
809     f = asksaveasfile(initialdir = self._DIR['saved_configurations'],
810                         title='choose your file name',
811                         filetypes=[('JSON files', '*.json')],
812                         defaultextension = 'json')
813
814     cfg = dict({'solarfield' : {}})
815     cfg['solarfield'].update(dict({'name' : self.enname.get()}))
816     cfg['solarfield'].update(dict({'rated_tin' : self.enratedtin.get()}))
817     cfg['solarfield'].update(dict({'rated_tout' : self.enratedtout.get()}))
818     cfg['solarfield'].update(dict({'rated_pin' : self.enratedpin.get()}))
819     cfg['solarfield'].update(dict({'rated_pout' : self.enratedpout.get()}))
820     cfg['solarfield'].update(dict({'rated_massflow' :
821         self.enratedmassflow.get()}))
822     cfg['solarfield'].update(dict({'min_massflow' :
823         self.enrecirculationmassflow.get()}))
824     cfg['solarfield'].update(dict({'tmin' : self.entmin.get()}))
825     cfg['solarfield'].update(dict({'tmax' : self.entmax.get()}))
826     cfg['solarfield'].update(dict({'loop': dict({'scas': self.enscas.get(),
827                                         'hces': self.enhces.get()})}))
828
829
830     subfields = []
831
832     for r in datarow:
833         sf = {}
834         index = 0
835         for v in r:
836             k = dictkeys[index]
837             sf[k]= self.to_number(v)
838             index += 1

```

```

839         subfields.append(sf)
840
841     cfg['solarfield'].update({'subfields' : subfields})
842     # cfg_settings['solarfield'].update(dict(cfg))
843     f.write(json.dumps(cfg))
844     f.close()
845
846 def showTagsTable(self):
847
848     self.msg = tk.Tk()
849     #self.fr_tags = tk.Frame(self.msg, )
850     self.strtags = tk.StringVar(self.msg)
851
852     for row in self.tagslist:
853         self.strtags.set(self.strtags.get() + '# ' +
854                         str(row[0]) + ' ---> ' +
855                         str(row[1]) + '\n')
856
857     self.lbtagslist = ttk.Label(self.msg,
858                                 textvariable =self.strtags).pack()
859
860     self.msg.mainloop()
861
862 def openTagsWizard(self):
863
864     tags_table = []
865
866     for tag in self.tagslist:
867         tags_table.append(['', tag])
868
869     columns_names = []
870
871     if self.varsimdatatype.get() == 2: # Data from field data file
872
873         columns_names.append(['DNI', '', ''])
874         columns_names.append(['Wspd', '', ''])
875         columns_names.append(['DryBulb', '', ''])
876         columns_names.append(['Pressure', '', ''])
877         columns_names.append(['GrossPower', '', ''])
878         columns_names.append(['AuxPower', '', ''])
879         columns_names.append(['NetPower', '', ''])
880
881     if self.varbenchmark.get():
882
883         for row in self.solarfield_table.table_data:
884             columns_names.append(['SB.'+row[0]+'.a.mf', '', ''])
885             columns_names.append(['SB.'+row[0]+'.a.tin', '', ''])
886             columns_names.append(['SB.'+row[0]+'.a.tout', '', ''])
887             columns_names.append(['SB.'+row[0]+'.a.pin', '', ''])
888             columns_names.append(['SB.'+row[0]+'.a.pout', '', ''])
889
890     self.columns_table.table_data = columns_names
891
892     self.showTagsTable()
893
894 def loadSelectedTags(self):
895
896     index = 0
897     for row in self.columns_table.table_data:
898         tag_index = self.to_number(row[1])

```

```

899     if  isinstance(tag_index, int):
900         new_row=[row[0], row[1], self.tagslist[tag_index-1][1]]
901         self.columns_table.update_row(
902             index,new_row)
903     index += 1
904
905 def buildSolarFieldFrame(self):
906
907     self.varsolarfieldname = tk.StringVar(self.fr_solarfield)
908     self.lbname = ttk.Label(self.fr_solarfield, text ='Solar Field Name' )
909     self.lbname.grid(row=0, column=0, sticky='W', padx=5, pady=5)
910     self.enname = ttk.Entry(self.fr_solarfield,
textvariable=self.varsolarfieldname)
911     self.enname.grid(row=0, column=1, sticky='W', padx=5, pady=5)
912
913     self.vartin = tk.DoubleVar(self.fr_solarfield)
914     self.lbratedtin = ttk.Label(self.fr_solarfield, text='Rated Tin [K]')
915     self.lbratedtin.grid(row=1, column=0, sticky='W', padx=5, pady=5)
916     self.enratedtin = ttk.Entry(self.fr_solarfield, textvariable=self.vartin)
917     self.enratedtin.grid(row=1, column=1, sticky='W', padx=5, pady=5)
918
919     self.vartout = tk.DoubleVar(self.fr_solarfield)
920     self.lbratedtout = ttk.Label(self.fr_solarfield, text='Rated Tout [K]')
921     self.lbratedtout.grid(row=1, column=2, sticky='W', padx=5, pady=5)
922     self.enratedtout = ttk.Entry(self.fr_solarfield, textvariable=self.vartout)
923     self.enratedtout.grid(row=1, column=3, sticky='W', padx=5, pady=5)
924
925     self.varpin = tk.DoubleVar(self.fr_solarfield)
926     self.lbratedpin = ttk.Label(self.fr_solarfield, text='Rated Pin [Pa]')
927     self.lbratedpin.grid(row=2, column=0, sticky='W', padx=5, pady=5)
928     self.enratedpin = ttk.Entry(self.fr_solarfield, textvariable=self.varpin)
929     self.enratedpin.grid(row=2, column=1, sticky='W', padx=5, pady=5)
930
931     self.varpout = tk.DoubleVar(self.fr_solarfield)
932     self.lbratedpout = ttk.Label(self.fr_solarfield, text='Rated Pout [Pa]')
933     self.lbratedpout.grid(row=2, column=2, sticky='W', padx=5, pady=5)
934     self.enratedpout = ttk.Entry(self.fr_solarfield, textvariable=self.varpout)
935     self.enratedpout.grid(row=2, column=3, sticky='W', padx=5, pady=5)
936
937     self.vartmin = tk.DoubleVar(self.fr_solarfield)
938     self.lbtmin = ttk.Label(self.fr_solarfield, text='Tmin [K]')
939     self.lbtmin.grid(row=3, column=0, sticky='W', padx=5, pady=5)
940     self.entmin = ttk.Entry(self.fr_solarfield, textvariable=self.vartmin)
941     self.entmin.grid(row=3, column=1, sticky='W', padx=5, pady=5)
942
943     self.vartmax = tk.DoubleVar(self.fr_solarfield)
944     self.lbtmax = ttk.Label(self.fr_solarfield, text='Tmax [K]')
945     self.lbtmax.grid(row=3, column=2, sticky='W', padx=5, pady=5)
946     self.entmax = ttk.Entry(self.fr_solarfield, textvariable=self.vartmax)
947     self.entmax.grid(row=3, column=3, sticky='W', padx=5, pady=5)
948
949     self.varratedmassflow = tk.DoubleVar(self.fr_solarfield)
950     self.lbratedmassflow = ttk.Label(
951         self.fr_solarfield, text='Loop Rated massflow [kg/s]')
952     self.lbratedmassflow.grid(row=4, column=0, sticky='W', padx=5, pady=5)
953     self.enratedmassflow = ttk.Entry(
954         self.fr_solarfield, textvariable=self.varratedmassflow)
955     self.enratedmassflow.grid(row=4, column=1, sticky='W', padx=5, pady=5)
956
957     self.varrecirculation = tk.DoubleVar(self.fr_solarfield)

```

```

958     self.lbrecirculationmassflow = ttk.Label(
959         self.fr_solarfield, text='Loop minimum massflow [kg/s]')
960     self.lbrecirculationmassflow.grid(
961         row=4, column=2, sticky='W', padx=5, pady=5)
962     self.enrecirculationmassflow = ttk.Entry(self.fr_solarfield,
963         textvariable=self.varrecirculation)
964     self.enrecirculationmassflow.grid(
965         row=4, column=3, sticky='W', padx=5, pady=5)
966
967     self.varrowspacing = tk.DoubleVar(self.fr_solarfield)
968     self.lbrowspacing = ttk.Label(
969         self.fr_solarfield, text='Row Spacing [m]')
970     self.lbrowspacing.grid(row=5, column=0, sticky='W', padx=5, pady=5)
971     self.enrowspacing = ttk.Entry(
972         self.fr_solarfield, textvariable=self.varrowspacing)
973     self.enrowspacing.grid(row=5, column=1, sticky='W', padx=5, pady=5)
974
975     self.varscattrackingtype = tk.IntVar(self.fr_solarfield)
976
977     self.lbscattrackingtype = ttk.Label(
978         self.fr_solarfield,
979         text='Tracking Axis (1: N-S, 2: E-W)').grid(
980             row=5, column=2, sticky='W', padx=2, pady=5)
981
982 # RadioButton for Tracking Type (Tracking Axis 1: N-S, 2: E-W)
983     self.rbtrackingtype = tk.Radiobutton(
984         self.fr_solarfield,
985         padx=2,
986         text = 'Tracking Axis N-S',
987         variable= self.varscattrackingtype, value=1).grid(
988             row=5, column=3, sticky='W', padx=2, pady=5)
989
990 # RadioButton for Tracking Type (Tracking Axis 1: N-S, 2: E-W)
991     self.rbtrackingtype = tk.Radiobutton(
992         self.fr_solarfield,
993         padx=2,
994         text = 'Tracking Axis E-W',
995         variable= self.varscattrackingtype, value=2).grid(
996             row=5, column=4, sticky='W', padx=2, pady=5)
997
998     self.varscas = tk.IntVar(self.fr_solarfield)
999     self.lbscas = ttk.Label(self.fr_solarfield, text='SCAs per Loop')
1000    self.lbscas.grid(row=6, column=0, sticky='W', padx=5, pady=5)
1001    self.enscas = ttk.Entry(self.fr_solarfield, textvariable=self.varscas)
1002    self.enscas.grid(row=6, column=1, sticky='W', padx=5, pady=5)
1003
1004    self.varhces = tk.IntVar(self.fr_solarfield)
1005    self.lbhces = ttk.Label(self.fr_solarfield, text='HCEs per SCA')
1006    self.lbhces.grid(row=6, column=2, sticky='W', padx=5, pady=5)
1007    self.enhces = ttk.Entry(self.fr_solarfield, textvariable=self.varhces)
1008    self.enhces.bind('<Key>', lambda event: self.updateHCEperSCA())
1009    self.enhces.grid(row=6, column=3, sticky='W', padx=5, pady=5)
1010
1011    self.solarfield_table = table.Tk_Table(
1012        self.fr_solarfield,
1013        ['SUBFIELD NAME', 'NUMBER OF LOOPS'],
1014        row_numbers=True,
1015        stripped_rows=('white', '#f2f2f2'),
1016        select_mode='none',
1017        cell_anchor='center',

```

```

1017         adjust_heading_to_content=True)
1018     self.solarfield_table.grid(
1019         row=7, column=0, columnspan =2, padx=5, pady=5)
1020
1021     self.columns_table = table.Tk_Table(
1022         self.fr_solarfield,
1023         [ ' COLUMN      ', ' TAG #' , '          TAG        ' ],
1024         row_numbers=True,
1025         stripped_rows=('white', '#f2f2f2'),
1026         select_mode='none',
1027         cell_anchor='center',
1028         adjust_heading_to_content=True)
1029     self.columns_table.grid(
1030         row=7, column=2, columnspan=2, padx=5, pady=5)
1031
1032     self.frame0 = ttk.Frame(self.fr_solarfield)
1033     self.frame0.grid(row=8, column=0, columnspan=2, padx=2, pady=5)
1034
1035     self.btnewrow = ttk.Button(
1036         self.frame0,
1037         text='Insert',
1038         command=lambda: self.__insert_rows__(self.solarfield_table))
1039     self.btnewrow.pack(side=tk.LEFT)
1040
1041     self.btdelrows = ttk.Button(
1042         self.frame0,
1043         text='Delete',
1044         command=lambda: self.__del_rows__(self.solarfield_table))
1045     self.btdelrows.pack(side=tk.LEFT)
1046
1047     self.frame1 = ttk.Frame(self.fr_solarfield)
1048     self.frame1.grid(row=8, column=2, columnspan=2, padx=2, pady=5)
1049
1050     self.bttagswizard = ttk.Button(
1051         self.frame1,
1052         text='Open TAGS wizard',
1053         command=lambda: self.openTagsWizard())
1054     self.bttagswizard.pack(side=tk.LEFT)
1055
1056     self.btloadtags = ttk.Button(
1057         self.frame1,
1058         text='Load TAGs',
1059         command=lambda: self.loadSelectedTags())
1060     self.btloadtags.pack(side=tk.LEFT)
1061
1062 # Site & Weather contruction Tab
1063 def dataLoadDialog(self, title, labeltext=''):
1064
1065     if self.varsimdatatype.get() == 1:
1066         path = askopenfilename(
1067             initialdir=self._DIR['weather_files'],
1068             title='choose your file',
1069             filetypes=(( 'TMY files' , ' *.tm2 '),
1070                         ( 'TMY files' , ' *.tm3 '),
1071                         ( 'csv files' , ' *.csv '),
1072                         ( 'all files' , ' *.* ')))
1073
1074         self.vardatafilepath.set(os.path.dirname(path)+ '/')
1075         self.vardatafilename.set(os.path.basename(path))
1076         self.vardatafileurl.set(os.path.dirname(path)+ '/' +

```

```

1077                               os.path.basename(path))
1078         elif self.varsimdatatype.get() == 2:
1079             path = askopenfilename(
1080                 initialdir=self._DIR['fielddata_files'],
1081                 title='choose your file',
1082                 filetypes=(( 'csv files', '*.csv'),
1083                             ('all files', '*.*')))
1084
1085             self.vardatafilepath.set(os.path.dirname(path)+'/')
1086             self.vardatafilename.set(os.path.basename(path))
1087             self.vardatafileurl.set(os.path.dirname(path)+'/'+os.path.basename(path))
1088
1089             df = pd.read_csv(path, sep=';',
1090                               decimal=',',
1091                               dayfirst=True,
1092                               index_col=0)
1093
1094             count = 0
1095             for r in list(df):
1096                 count += 1
1097                 self.tagslist.append([count, r])
1098             else:
1099                 tk.messagebox.showwarning(
1100                     title='Warning',
1101                     message='Check Data Source Selection')
1102
1103         if self.varloadsitedata.get():
1104
1105             strfilename, strext = os.path.splitext(path)
1106             if strext == '.csv':
1107                 weatherdata = pvlib.iotools.tmy.read_tmy3(path)
1108                 file = path
1109             elif (strext == '.tm2' or strext == '.tmy'):
1110                 weatherdata = pvlib.iotools.tmy.read_tmy2(path)
1111                 file = path
1112             elif strext == '.xls':
1113                 pass
1114             else:
1115                 print('unknow extension ', strext)
1116                 return
1117
1118             self.varsitename.set(weatherdata[1]['City'])
1119             self.varsitelat.set(weatherdata[1]['latitude'])
1120             self.varsitelong.set(weatherdata[1]['longitude'])
1121             self.varsitealt.set(weatherdata[1]['altitude'])
1122
1123         def select_fluid_from_csv(self):
1124
1125             # abre una ventana de selección de csv.
1126             self.loadWindow= tk.Tk()
1127             self.loadWindow.attributes('-fullscreen', False)
1128             self.loadWindow.title('Selection Window')
1129
1130             # Menu
1131             self.loadWindow.menubar = tk.Menu(
1132                 self.loadWindow)
1133             self.loadWindow.load_menu = tk.Menu(
1134                 self.loadWindow.menubar, tearoff=0)
1135             self.loadWindow.load_menu.add_command(
1136                 label='Open', command=self.open_csv())
1137             self.loadWindow.load_menu.add_separator()

```

```

1137     self.loadWindow.load_menu.add_command(
1138         label='Exit', command=None)
1139     self.loadWindow.menuBar.add_cascade(
1140         label='Select File', menu=self.loadWindow.load_menu)
1141
1142     self.loadWindow.config(menu=self.loadWindow.menuBar)
1143
1144 def open_csv(self, path=None):
1145
1146     df = pd.DataFrame()
1147
1148     try:
1149         if path is None:
1150             root = Tk()
1151             root.withdraw()
1152             path = askopenfilename(initialdir = '.\fielddata_files\' ,
1153                                     title='choose your file',
1154                                     filetypes=(('csv files','*.csv'),
1155                                     ('all files','*.*')))
1156             root.update()
1157             root.destroy()
1158
1159         if path is None:
1160             return
1161         else:
1162             strfilename, strext = os.path.splitext(path)
1163
1164         if  strext == '.csv':
1165             print('csv.....')
1166
1167             df = pd.read_csv(path, sep=';',
1168                               decimal= ',')
1169             file = path
1170         else:
1171             print('unknow extension ', strext)
1172             return
1173     else:
1174         strfilename, strext = os.path.splitext(path)
1175
1176         if  strext == '.csv':
1177             print('csv...')
1178             df = pd.read_csv(path, sep=';',
1179                               decimal= ',')
1180             file = path
1181         else:
1182             print('unknow extension ', strext)
1183             return
1184
1185     except Exception:
1186         raise
1187
1188 def load_fluid_library(self):
1189
1190     path = askopenfilename(initialdir = self._DIR['fluid_files'],
1191                           title='choose your file',
1192                           filetypes=[('JSON files', '*.json')])'
1193
1194     with open(path) as cfg_file:
1195         cfg = json.load(cfg_file, parse_float= float, parse_int= int)
1196

```

```

1197     self.fluid_list = cfg
1198
1199     self.cmbfluidname['values'] = [s['name'] for s in cfg ]
1200     self.cmbfluidname.current(0)
1201     self.load_fluid_parameters()
1202
1203 def load_fluid_parameters(self):
1204
1205     self.fluid_config = {}
1206
1207     for item in self.fluid_list:
1208
1209         if item['name'] == self.cmbfluidname.get():
1210             self.fluid_config = item
1211
1212     datarow=[ ]
1213
1214     for parameter in self.fluid_config.keys():
1215         if parameter in ['cp','mu','rho','kt','h','t']:
1216             datarow.append([parameter] + self.fluid_config[parameter])
1217
1218     self.fluid_table.table_data = datarow
1219
1220     self.varfluidname.set(self.fluid_config['name'])
1221     self.varfluidtmin.set(self.fluid_config['tmin'])
1222     self.varfluidtmax.set(self.fluid_config['tmax'])
1223
1224 def fluid_load_dialog(self):
1225
1226     path = askopenfilename(initialdir=self._DIR['fluid_files'],
1227                           title='choose your file',
1228                           filetypes=[('JSON files', '*.json')])
1229
1230     with open(path) as cfg_file:
1231         cfg = json.load(cfg_file, parse_float=float, parse_int=int)
1232
1233     cp_coefs = [[[]]*10
1234     rho_coefs = [[[]]*10
1235     mu_coefs = [[[]]*10
1236     kt_coefs = [[[]]*10
1237     h_coefs = [[[]]*10
1238     t_coefs = [[[]]*10
1239
1240     temp_cp = ['cp']
1241     temp_rho = ['rho']
1242     temp_mu = ['mu']
1243     temp_kt = ['kt']
1244     temp_h = ['h']
1245     temp_t = ['t']
1246
1247     grades = ['Factor']
1248     grades.extend([0, 1, 2, 3, 4, 5, 6, 7, 8])
1249
1250     temp_cp.extend(list(cfg['hot_fluid']['cp']))
1251     temp_rho.extend(list(cfg['hot_fluid']['rho']))
1252     temp_mu.extend(list(cfg['hot_fluid']['mu']))
1253     temp_kt.extend(list(cfg['hot_fluid']['kt']))
1254     temp_h.extend(list(cfg['hot_fluid']['h']))
1255     temp_t.extend(list(cfg['hot_fluid']['t']))
1256

```

```

1257     for index in range(len(temp_cp)):
1258         cp_coefs[index] = temp_cp[index]
1259     cp_coefs[1:] = [Decimal(s) for s in cp_coefs[1:]]
1260     for index in range(len(temp_rho)):
1261         rho_coefs[index] = temp_rho[index]
1262     rho_coefs[1:] = [Decimal(s) for s in rho_coefs[1:]]
1263     for index in range(len(temp_mu)):
1264         mu_coefs[index] = temp_mu[index]
1265     mu_coefs[1:] = [Decimal(s) for s in mu_coefs[1:]]
1266     for index in range(len(temp_kt)):
1267         kt_coefs[index] = temp_kt[index]
1268     kt_coefs[1:] = [Decimal(s) for s in kt_coefs[1:]]
1269     for index in range(len(temp_h)):
1270         h_coefs[index] = temp_h[index]
1271     h_coefs[1:] = [Decimal(s) for s in h_coefs[1:]]
1272     for index in range(len(temp_t)):
1273         t_coefs[index] = temp_t[index]
1274     t_coefs[1:] = [Decimal(s) for s in t_coefs[1:]]
1275
1276     datarow = []
1277     datarow.append(cp_coefs)
1278     datarow.append(rho_coefs)
1279     datarow.append(mu_coefs)
1280     datarow.append(kt_coefs)
1281     datarow.append(h_coefs)
1282     datarow.append(t_coefs)
1283
1284     self.fluid_table.table_data = datarow
1285     self.varfluidtmin.set(cfg['tmin'])
1286     self.varfluidtmax.set(cfg['tmax'])
1287     self.varfluidname.set(cfg['name'])
1288
1289     self.fr_fluid.update()
1290
1291 def checkfluid(self):
1292
1293     var = self.varfluidtable.get()
1294
1295     if var == 1: # Fluid from table
1296         self.cmbcoolpropID.set('')
1297         self.cmbcoolpropID['state'] = 'disabled'
1298         self.enfluidname['state'] = 'normal'
1299         self.btloadfluidcfg['state'] = 'normal'
1300         self.enfluidtmin['state'] = 'normal'
1301         self.enfluidtmax['state'] = 'normal'
1302         self.varfluidtmax.set('')
1303         self.varfluidtmin.set('')
1304         self.varcoolproptmin.set('')
1305         self.varcoolproptmax.set('')
1306     elif var == 2: # Fluid from CoolProp
1307         self.varcoolproptmax.set('')
1308         self.varcoolproptmin.set('')
1309         self.varfluidname.set('')
1310         self.cmbcoolpropID['state'] = 'readonly'
1311         self.enfluidname['state'] = 'disabled'
1312         self.btloadfluidcfg['state'] = 'disabled'
1313         self.enfluidtmin['state'] = 'disabled'
1314         self.enfluidtmax['state'] = 'disabled'
1315         self.varfluidtmax.set('')
1316         self.varfluidtmin.set('')

```

```

1317         self.varcoolproptmin.set('')
1318         self.varcoolproptmax.set('')
1319         self.fluid_table.table_data = []
1320     else:
1321         pass
1322
1323     def get_coolprop_data(self):
1324
1325         self.varcoolproptmin.set(CP.PropsSI("TMIN", self.cmbcoolpropID.get()))
1326         self.varcoolproptmax.set(CP.PropsSI("TMAX", self.cmbcoolpropID.get()))
1327
1328
1329     def buildFluidFrame(self):
1330
1331         self.varfluidtmin = tk.DoubleVar(self.fr_fluid)
1332         self.varfluidtmax = tk.DoubleVar(self.fr_fluid)
1333         self.varcoolproptmin = tk.DoubleVar(self.fr_fluid)
1334         self.varcoolproptmax = tk.DoubleVar(self.fr_fluid)
1335         self.varfluidname = tk.StringVar(self.fr_fluid)
1336         self.fluid_list = []
1337         self.varfluidtable = tk.IntVar(self.fr_fluid)
1338
1339         self.varfluidtable.set(1) # 1 for Table, 2 for CoolProp Library
1340
1341         # RadioButton for fluid from library
1342         self.rbfliuidlib = tk.Radiobutton(
1343             self.fr_fluid,
1344             padx=1,
1345             text='Fluid Data from CoolProp',
1346             variable=self.varfluidtable, value=2,
1347             command=lambda: self.checkfluid()).grid(
1348                 row=0, column=0, sticky='W', padx=1, pady=5)
1349
1350         self.lbcoolpropID = tk.Label(
1351             self.fr_fluid, text='CoolProp ID (INCOMP::xxxx)')
1352         self.lbcoolpropID.grid(row=0, column=1, sticky='W', padx=1, pady=5)
1353
1354         self.cmbcoolpropID = ttk.Combobox(self.fr_fluid)
1355         self.cmbcoolpropID.bind(
1356             "<<ComboboxSelected>>", lambda event: self.get_coolprop_data())
1357         self.cmbcoolpropID['values'] = self._COOLPROP_FLUIDS
1358         self.cmbcoolpropID['state'] = 'readonly'
1359         self.cmbcoolpropID.current(0)
1360         self.cmbcoolpropID.grid(row=0, column=2, sticky='W', padx=1, pady=5)
1361
1362         self.lbcoolproptmintext = tk.Label(
1363             self.fr_fluid,
1364             text='Tmin[K]')
1365         self.lbcoolproptmintext.grid(
1366             row=0, column=3, sticky='E', padx=1, pady=5)
1367
1368         self.lbcoolproptmin = tk.Label(
1369             self.fr_fluid,
1370             textvariable=self.varcoolproptmin)
1371         self.lbcoolproptmin.grid(
1372             row=0, column=4, sticky='E', padx=1, pady=5)
1373
1374         self.lbcoolproptmaxtext = tk.Label(
1375             self.fr_fluid,
1376             text='Tmax[K]')

```

```

1377     self.lbcoolproptmaxtext.grid(
1378         row=0, column=5, sticky='W', padx=1, pady=5)
1379
1380     self.lbcoolproptmax = tk.Label(
1381         self.fr_fluid,
1382         textvariable=self.varcoolproptmax)
1383     self.lbcoolproptmax.grid(
1384         row=0, column=6, sticky='W', padx=1, pady=5)
1385
1386
1387     self.separator = ttk.Separator(self.fr_fluid).grid(
1388         row=1, column=0, columnspan=99, sticky=(tk.W, tk.E))
1389
1390     # RadioButton for fluid from table
1391     self.rbfluidtable = tk.Radiobutton(
1392         self.fr_fluid,
1393         padx=2,
1394         text = 'Fluid Data from table',
1395         variable=self.varfluidtable, value=1,
1396         command=lambda: self.checkfluid()).grid(
1397             row=2, column=0, columnspan=2, sticky='W', padx=2, pady=5)
1398
1399     self.lbfluidname = ttk.Label(self.fr_fluid, text='Name')
1400     self.lbfluidname.grid(row=3, column=0, sticky='W', padx=2, pady=5)
1401     self.enfluidname = ttk.Entry(self.fr_fluid, textvariable=self.varfluidname)
1402     self.enfluidname.grid(row=3, column=1, sticky='W', padx=2, pady=5)
1403
1404     self.cmbfluidname = ttk.Combobox(self.fr_fluid)
1405     self.cmbfluidname.bind('<<ComboboxSelected>>',
1406                             lambda event: self.load_fluid_parameters())
1407     self.cmbfluidname['values'] = self.fluid_list
1408     self.cmbfluidname.grid(row=3, column=2, padx=5, pady=5)
1409
1410     self.btloadfluidcfg = ttk.Button(
1411         self.fr_fluid, text='Load config',
1412         command=lambda: self.load_fluid_library())
1413     self.btloadfluidcfg.grid(
1414         row=3, column=3, sticky='W', padx=2, pady=5)
1415
1416     self.lbfluidtmin = ttk.Label(self.fr_fluid, text='Tmin[K]')
1417     self.lbfluidtmin.grid(row=4, column=0, sticky='W', padx=2, pady=5)
1418     self.enfluidtmin = ttk.Entry(
1419         self.fr_fluid, textvariable=self.varfluidtmin)
1420     self.enfluidtmin.grid(row=4, column=1, sticky='W', padx=2, pady=5)
1421
1422     self.lbfluidtmax = ttk.Label(self.fr_fluid, text='Tmax [K]')
1423     self.lbfluidtmax.grid(row=4, column=2, sticky='W', padx=2, pady=5)
1424     self.enfluidtmax = ttk.Entry(self.fr_fluid, textvariable=self.varfluidtmax)
1425     self.enfluidtmax.grid(row=4, column=3, sticky='W', padx=2, pady=5)
1426
1427     self.fluid_table = table.Tk_Table(
1428         self.fr_fluid,
1429         ['Parameter',
1430          'A x^0', 'B x^1',
1431          'C x^2', 'D x^3',
1432          'E x^4', 'F x^5',
1433          'G x^6', 'H x^7',
1434          'I x^8'],
1435         row_numbers=True,
1436         stripped_rows=('white', '#f2f2f2'),

```

```

1437         select_mode='none',
1438         cell_anchor='e',
1439         adjust_heading_to_content=True)
1440     self.fluid_table.grid(
1441         row=5, column=0, columnspan=7, sticky='W', padx=2, pady=5)
1442
1443     self.checkfluid()
1444
1445 # SCA Construction tab
1446 def load_sca_library(self):
1447
1448     path = askopenfilename(initialdir=self._DIR['sca_files'],
1449                           title='choose your file',
1450                           filetypes=[('JSON files', '*.json')])
1451
1452     with open(path) as cfg_file:
1453         cfg = json.load(cfg_file, parse_float= float, parse_int= int)
1454
1455     self.sca_list = cfg
1456     self.cmbscaname['values'] = [s['Name'] for s in cfg]
1457     self.cmbscaname.current(0)
1458     self.load_sca_parameters()
1459     self.checkfluid()
1460
1461
1462 def load_sca_parameters(self):
1463
1464     self.sca_config = {}
1465
1466     for item in self.sca_list:
1467         if item['Name'] == self.cmbscaname.get():
1468             self.sca_config = item
1469
1470         self.varscaname.set(self.sca_config['Name'])
1471         self.varscalength.set(self.sca_config['SCA Length'])
1472         self.varscaaperture.set(self.sca_config['Aperture'])
1473         self.varscafocallen.set(self.sca_config['Focal Len'])
1474         self.varscIAMF0.set(self.sca_config['IAM Coefficient F0'])
1475         self.varscIAMF1.set(self.sca_config['IAM Coefficient F1'])
1476         self.varscIAMF2.set(self.sca_config['IAM Coefficient F2'])
1477         self.varscareflectance.set(self.sca_config['Reflectance'])
1478         self.varscageoaccuracy.set(self.sca_config['Geom.Accuracy'])
1479         self.varscatracktwist.set(self.sca_config['Track Twist'])
1480         self.varscacleanliness.set(self.sca_config['Cleanliness'])
1481         self.varscafactor.set(self.sca_config['Factor'])
1482         self.varscavailability.set(self.sca_config['Availability'])
1483         self.updateHCEperSCA()
1484
1485 def buildSCAFrame(self):
1486
1487     self.sca_list = []
1488
1489     self.varscaname = tk.StringVar(self.fr_sca)
1490     self.varscalength = tk.DoubleVar(self.fr_sca)
1491     self.varscaaperture = tk.DoubleVar(self.fr_sca)
1492     self.varscafocallen = tk.DoubleVar(self.fr_sca)
1493     self.varscIAMF0 = tk.DoubleVar(self.fr_sca)
1494     self.varscIAMF1 = tk.DoubleVar(self.fr_sca)
1495     self.varscIAMF2 = tk.DoubleVar(self.fr_sca)
1496     self.varscareflectance = tk.DoubleVar(self.fr_sca)

```

```
1497     self.varscageoaccuracy = tk.DoubleVar(self.fr_sca)
1498     self.varscatracktwist = tk.DoubleVar(self.fr_sca)
1499     self.varscacleanliness = tk.DoubleVar(self.fr_sca)
1500     self.varscafactor = tk.DoubleVar(self.fr_sca)
1501     self.varscaavailability = tk.DoubleVar(self.fr_sca)
1502
1503     self.lbscaname = ttk.Label(
1504         self.fr_sca,
1505         text='SCA base name').grid(
1506             row=0, column=0, sticky='W', padx=2, pady=5)
1507
1508     self.enscaname = ttk.Entry(
1509         self.fr_sca,
1510         textvariable=self.varscaavailability).grid(
1511             row=0, column=1, sticky='W', padx=2, pady=5)
1512
1513     self.btscaload = ttk.Button(
1514         self.fr_sca, text='Load Library',
1515         command=lambda: self.load_sca_library())
1516     self.btscaload.grid(row=0, column=7, sticky='W', padx=2, pady=5)
1517
1518     self.cmbscaname = ttk.Combobox(self.fr_sca)
1519     self.cmbscaname.bind(
1520         '<>ComboboxSelected>>', lambda event: self.load_sca_parameters())
1521     self.cmbscaname['values'] = self.sca_list
1522     self.cmbscaname.grid(row=0, column=3, columnspan=4, padx=5, pady=5)
1523
1524     self.lbscalength = ttk.Label(
1525         self.fr_sca,
1526         text='SCA Length [m]').grid(
1527             row=1, column=0, sticky='W', padx=2, pady=5)
1528     self.enscalength = ttk.Entry(
1529         self.fr_sca,
1530         textvariable=self.varscalength)
1531     self.enscalength.bind('<Key>', lambda event: self.updateHCEperSCA())
1532     self.enscalength.grid(
1533             row=1, column=1, sticky='W', padx=2, pady=5)
1534
1535     self.lbscaaperture = ttk.Label(
1536         self.fr_sca,
1537         text='SCA aperture [m]').grid(
1538             row=2, column=0, sticky='W', padx=2, pady=5)
1539     self.enscaaperture = ttk.Entry(
1540         self.fr_sca,
1541         textvariable=self.varscaaperture).grid(
1542             row=2, column=1, sticky='W', padx=2, pady=5)
1543
1544     self.lbscafocallen = ttk.Label(
1545         self.fr_sca,
1546         text='SCA focal length [m]').grid(
1547             row=3, column=0, sticky='W', padx=2, pady=5)
1548     self.enscafocallen = ttk.Entry(
1549         self.fr_sca,
1550         textvariable=self.varscafocallen).grid(
1551             row=3, column=1, sticky='W', padx=2, pady=5)
1552
1553     self.lbscaIAMF0 = ttk.Label(
1554         self.fr_sca,
1555         text='SCA IAM factor F0 []').grid(
1556             row=4, column=0, sticky='W', padx=2, pady=5)
```

```
1557     self.enscaIAMF0 = ttk.Entry(
1558         self.fr_sca,
1559         textvariable=self.varscaIAMF0).grid(
1560             row=4, column=1, sticky='W', padx=2, pady=5)
1561
1562     self.lbscaIAMF1 = ttk.Label(
1563         self.fr_sca,
1564         text='SCA IAM factor F1 []').grid(
1565             row=5, column=0, sticky='W', padx=2, pady=5)
1566     self.enscaIAMF1 = ttk.Entry(
1567         self.fr_sca,
1568         textvariable=self.varscaIAMF1).grid(
1569             row=5, column=1, sticky='W', padx=2, pady=5)
1570
1571     self.lbscaIAMF2 = ttk.Label(
1572         self.fr_sca,
1573         text='SCA IAM factor F2 []').grid(
1574             row=6, column=0, sticky='W', padx=2, pady=5)
1575     self.enscaIAMF2 = ttk.Entry(
1576         self.fr_sca,
1577         textvariable=self.varscaIAMF2).grid(
1578             row=6, column=1, sticky='W', padx=2, pady=5)
1579
1580     self.lbscareflectance = ttk.Label(
1581         self.fr_sca,
1582         text='SCA mirror reflectance []').grid(
1583             row=7, column=0, sticky='W', padx=2, pady=5)
1584     self.enscareflectance = ttk.Entry(
1585         self.fr_sca,
1586         textvariable=self.varscareflectance).grid(
1587             row=7, column=1, sticky='W', padx=2, pady=5)
1588
1589     self.lbscageoaccuracy = ttk.Label(
1590         self.fr_sca,
1591         text='SCA geometric accuracy []').grid(
1592             row=8, column=0, sticky='W', padx=2, pady=5)
1593     self.enscageoaccuracy = ttk.Entry(
1594         self.fr_sca,
1595         textvariable=self.varscageoaccuracy).grid(
1596             row=8, column=1, sticky='W', padx=2, pady=5)
1597
1598     self.lbscatracketwist = ttk.Label(
1599         self.fr_sca,
1600         text='SCA Tracking Twist []').grid(
1601             row=9, column=0, sticky='W', padx=2, pady=5)
1602     self.enscatracketwist = ttk.Entry(
1603         self.fr_sca,
1604         textvariable=self.varscatracketwist).grid(
1605             row=9, column=1, sticky='W', padx=2, pady=5)
1606
1607     self.lbscacleanliness = ttk.Label(
1608         self.fr_sca,
1609         text='SCA Cleanliness []').grid(
1610             row=10, column=0, sticky='W', padx=2, pady=5)
1611     self.enscacleanliness = ttk.Entry(
1612         self.fr_sca,
1613         textvariable=self.varscacleanliness).grid(
1614             row=10, column=1, sticky='W', padx=2, pady=5)
1615
1616     self.lbscafactor = ttk.Label(
```

```

1617     self.fr_sca,
1618     text='SCA Factor []').grid(
1619         row=11, column=0, sticky='W', padx=2, pady=5)
1620 self.enscafactor = ttk.Entry(
1621     self.fr_sca,
1622     textvariable=self.varscafactor).grid(
1623         row=11, column=1, sticky='W', padx=2, pady=5)
1624
1625     self.lbscaavailability = ttk.Label(
1626         self.fr_sca,
1627         text='SCA Availability []').grid(
1628             row=12, column=0, sticky='W', padx=2, pady=5)
1629 self.ensaavailability = ttk.Entry(
1630     self.fr_sca,
1631     textvariable=self.varscaavailability).grid(
1632         row=12, column=1, sticky='W', padx=2, pady=5)
1633
1634 def load_hce_parameters(self):
1635
1636     self.hce_config = {}
1637
1638     for item in self.hce_list:
1639
1640         if item['Name'] == self.cmbhcename.get():
1641             self.hce_config = item
1642
1643             self.varhcename.set(self.hce_config['Name'])
1644             self.varhcedri.set(self.hce_config['Absorber tube inner diameter'])
1645             self.varhcedro.set(self.hce_config['Absorber tube outer diameter'])
1646             self.varhcedgi.set(self.hce_config['Glass envelope inner diameter'])
1647             self.varhcedgo.set(self.hce_config['Glass envelope outer diameter'])
1648             self.varhcelength.set(self.hce_config['Length'])
1649             self.varhceinnerroughness.set(self.hce_config['Inner surface roughness'])
1650             self.varhceminreynolds.set(self.hce_config['Min Reynolds'])
1651             self.varhceemittanceA0.set(self.hce_config['Absorber emittance factor A0'])
1652             self.varhceemittanceA1.set(self.hce_config['Absorber emittance factor A1'])
1653             self.varhceabsorptance.set(self.hce_config['Absorber absorptance'])
1654             self.varhcetransmittance.set(self.hce_config['Envelope transmittance'])
1655             self.varhcebrackets.set(self.hce_config['Brackets'])
1656             self.updateHCEperSCA()
1657
1658 def load_hce_library(self, path = None):
1659
1660     if path is None:
1661         path = askopenfilename(initialdir = self._DIR['hce_files'],
1662                             title='choose your file',
1663                             filetypes=[('JSON files', '*.json')])
1664
1665     with open(path) as cfg_file:
1666         cfg = json.load(cfg_file, parse_float= float, parse_int= int)
1667
1668         self.hce_list = cfg
1669         self.cmbhcename['values'] = [s['Name'] for s in cfg ]
1670         self.cmbhcename.current(0)
1671         self.load_hce_parameters()
1672
1673
1674 def updateHCEperSCA(self):
1675
1676     var1 = self.varscalelength.get()

```

```

1677     var2 = self.varhcelength.get()
1678
1679     if var2 != 0:
1680         self.varhcepersca.set(var1 // var2)
1681     else:
1682         self.varhcepersca.set(0.0)
1683
1684     var3 = self.varhcepersca.get()
1685     var4 = self.varhces.get()
1686
1687     if var3 >= var4:
1688         self.varhceperscatext.set('Max. ' + str(var3) + ' HCE per SCA')
1689     else:
1690         self.varhceperscatext.set(
1691             'Error: ' + str(var4) +
1692             ' exceeded max. HCE per SCA = ' + str(var3))
1693
1694 def buildHCEFrame(self):
1695
1696     self.hce_list = []
1697     self.varhcename = tk.StringVar(self.fr_hce)
1698     self.varhcedri = tk.DoubleVar(self.fr_hce)
1699     self.varhcedro = tk.DoubleVar(self.fr_hce)
1700     self.varhcedgi = tk.DoubleVar(self.fr_hce)
1701     self.varhcedgo = tk.DoubleVar(self.fr_hce)
1702     self.varhcelength = tk.DoubleVar(self.fr_hce)
1703     self.varhceinnerroughness = tk.DoubleVar(self.fr_hce)
1704     self.varhceminreynolds = tk.DoubleVar(self.fr_hce)
1705     self.varhcepersca = tk.DoubleVar(self.fr_hce)
1706     self.varhceperscatext = tk.StringVar(self.fr_hce)
1707     self.varhceabsorptance = tk.DoubleVar(self.fr_hce)
1708     self.varhcetransmittance = tk.DoubleVar(self.fr_hce)
1709     self.varhceemittanceA0 = tk.DoubleVar(self.fr_hce)
1710     self.varhceemittanceA1 = tk.DoubleVar(self.fr_hce)
1711     self.varbellowsratio = tk.DoubleVar(self.fr_hce)
1712     self.varshieldshading = tk.DoubleVar(self.fr_hce)
1713     self.varhcebrackets = tk.DoubleVar(self.fr_hce)
1714
1715     self.lbhcename = ttk.Label(
1716         self.fr_hce,
1717         text='HCE base name').grid(
1718             row=0, column=0, sticky='W', padx=2, pady=5)
1719     self.enhcename = ttk.Entry(
1720         self.fr_hce,
1721         textvariable=self.varhcename).grid(
1722             row=0, column=1, sticky='W', padx=2, pady=5)
1723
1724     self.cmbhcename = ttk.Combobox(self.fr_hce)
1725     self.cmbhcename.bind('<<ComboboxSelected>>', lambda event:
1726         self.load_hce_parameters())
1727         self.cmbhcename['values'] = self.hce_list
1728         self.cmbhcename.grid(row=0, column=2, padx=5, pady=5)
1729
1730         self.bthceload = ttk.Button(
1731             self.fr_hce, text='Load Library',
1732             command=lambda: self.load_hce_library())
1733         self.bthceload.grid(row=0, column=3, sticky='W', padx=2, pady=5)
1734
1735         self.lbhcedri = ttk.Label(
1736             self.fr_hce,

```

```
1736     text='HCE inner diameter [m]').grid(
1737         row=1, column=0, sticky='W', padx=2, pady=5)
1738 self.enhcedri = ttk.Entry(
1739     self.fr_hce,
1740     textvariable=self.varhcedri).grid(
1741         row=1, column=1, sticky='W', padx=2, pady=5)
1742
1743 self.lbhcedro = ttk.Label(
1744     self.fr_hce,
1745     text='HCE outer diameter [m]').grid(
1746         row=2, column=0, sticky='W', padx=2, pady=5)
1747 self.enhcedro = ttk.Entry(
1748     self.fr_hce,
1749     textvariable=self.varhcedro).grid(
1750         row=2, column=1, sticky='W', padx=2, pady=5)
1751
1752 self.lbhcedgi = ttk.Label(
1753     self.fr_hce,
1754     text='HCE glass inner diameter [m]').grid(
1755         row=3, column=0, sticky='W', padx=2, pady=5)
1756 self.enhcedgi = ttk.Entry(
1757     self.fr_hce,
1758     textvariable=self.varhcedgi).grid(
1759         row=3, column=1, sticky='W', padx=2, pady=5)
1760
1761 self.lbhcedgo = ttk.Label(
1762     self.fr_hce,
1763     text='HCE glass outer diameter [m]').grid(
1764         row=4, column=0, sticky='W', padx=2, pady=5)
1765 self.enhcedgo = ttk.Entry(
1766     self.fr_hce,
1767     textvariable=self.varhcedgo).grid(
1768         row=4, column=1, sticky='W', padx=2, pady=5)
1769
1770 self.lbhcelong = ttk.Label(
1771     self.fr_hce,
1772     text='HCE longitude [m]').grid(
1773         row=5, column=0, sticky='W', padx=2, pady=5)
1774 self.enhcelong = ttk.Entry(
1775     self.fr_hce,
1776     textvariable=self.varhcelength)
1777 self.enhcelong.bind('<Key>', lambda event: self.updateHCEperSCA())
1778 self.enhcelong.grid(row=5, column=1, sticky='W', padx=2, pady=5)
1779
1780 self.lbhcepersca = ttk.Label(
1781     self.fr_hce,
1782     textvariable=self.varhceperscatext).grid(
1783         row=5, column=2, sticky='W', padx=2, pady=5)
1784
1785 self.lbbellowsratio = ttk.Label(
1786     self.fr_hce,
1787     text='Bellows ratio [ ]').grid(
1788         row=6, column=0, sticky='W', padx=2, pady=5)
1789 self.enbellowsratio = ttk.Entry(
1790     self.fr_hce,
1791     textvariable=self.varbellowsratio).grid(
1792         row=6, column=1, sticky='W', padx=2, pady=5)
1793
1794 self.lbshieldshading = ttk.Label(
1795     self.fr_hce,
```

```
1796     text='Shield shading [ ]').grid(
1797         row=7, column=0, sticky='W', padx=2, pady=5)
1798 self.enshieldshading = ttk.Entry(
1799     self.fr_hce,
1800     textvariable=self.varshieldshading).grid(
1801         row=7, column=1, sticky='W', padx=2, pady=5)
1802
1803 self.lbbrackets = ttk.Label(
1804     self.fr_hce,
1805     text='Brackets spacing').grid(
1806         row=8, column=0, sticky='W', padx=2, pady=5)
1807 self.enbrackets = ttk.Entry(
1808     self.fr_hce,
1809     textvariable=self.varhcebrackets).grid(
1810         row=8, column=1, sticky='W', padx=2, pady=5)
1811
1812 self.lbhceinnerroughness = ttk.Label(
1813     self.fr_hce,
1814     text='HCE inner roughness [ ]').grid(
1815         row=10, column=0, sticky='W', padx=2, pady=5)
1816 self.enhceinnerroughness = ttk.Entry(
1817     self.fr_hce,
1818     textvariable=self.varhceinnerroughness).grid(
1819         row=10, column=1, sticky='W', padx=2, pady=5)
1820
1821 self.lbhceminreynolds = ttk.Label(
1822     self.fr_hce,
1823     text='HCE minimum Reynolds Number [ ]').grid(
1824         row=11, column=0, sticky='W', padx=2, pady=5)
1825 self.enhceminreynolds = ttk.Entry(
1826     self.fr_hce,
1827     textvariable=self.varhceminreynolds).grid(
1828         row=11, column=1, sticky='W', padx=2, pady=5)
1829
1830 self.lbhceabsorptance = ttk.Label(
1831     self.fr_hce,
1832     text='Absorptance [ ]').grid(
1833         row=12, column=0, sticky='W', padx=2, pady=5)
1834 self.enhceabsorptance = ttk.Entry(
1835     self.fr_hce,
1836     textvariable=self.varhceabsorptance).grid(
1837         row=12, column=1, sticky='W', padx=2, pady=5)
1838
1839 self.lbhctransmittance = ttk.Label(
1840     self.fr_hce,
1841     text='Transmittance [ ]').grid(
1842         row=13, column=0, sticky='W', padx=2, pady=5)
1843 self.enhcetransmittance = ttk.Entry(
1844     self.fr_hce,
1845     textvariable=self.varhcetransmittance).grid(
1846         row=13, column=1, sticky='W', padx=2, pady=5)
1847
1848 self.lbemittanceA0 = ttk.Label(
1849     self.fr_hce,
1850     text='Emittance Factor A0 [ ]').grid(
1851         row=14, column=0, sticky='W', padx=2, pady=5)
1852 self.enemittanceA0 = ttk.Entry(
1853     self.fr_hce,
1854     textvariable=self.varhceemittanceA0).grid(
1855         row=14, column=1, sticky='W', padx=2, pady=5)
```

```
1856
1857     self.lbemittanceA1 = ttk.Label(
1858         self.fr_hce,
1859         text='Emittance Factor A1 [ ]').grid(
1860             row=15, column=0, sticky='W', padx=2, pady=5)
1861     self.enemittanceA1 = ttk.Entry(
1862         self.fr_hce,
1863         textvariable=self.varhceemittanceA1).grid(
1864             row=15, column=1, sticky='W', padx=2, pady=5)
1865
1866
1867 if __name__ == '__main__':
1868
1869     try:
1870         from Tkinter import Tk
1871         import tkMessageBox as messagebox
1872
1873     except ImportError:
1874         from tkinter import Tk
1875         from tkinter import messagebox
1876
1877     interface = Interface()
1878     interface.root.mainloop()
```