

# 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	FRANCISCO JOSÉ MUNUERA PÉREZ
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 por 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 . . . . .	2
1.3. Concentradores cilindroparabólicos . . . . .	2
1.3.1. El concentrador cilindroparabólico . . . . .	3
1.3.2. El tubo absorbedor o receptor . . . . .	5
1.3.3. El sistema de seguimiento . . . . .	6
1.3.4. El fluido caloportador . . . . .	7
2. DESCRIPCIÓN DE LOS MODELOS TEÓRICOS UTILIZADOS . . . . .	8
2.1. Modelo de rendimiento térmico de 4º Orden para caracterización de un sistema de captación solar . . . . .	8
2.2. Modelo de Primer Orden . . . . .	13
2.3. Modelo simplificado . . . . .	14
2.4. Aplicabilidad de los modelos . . . . .	15
3. MODELADO DEL CAMPO SOLAR . . . . .	17
3.1. Metodología seguida para el modelado del campo solar . . . . .	17
3.2. Sistemas físicos y Clases para el modelado del campo solar . . . . .	17
3.2.1. Clases derivadas de Model: Clase ModelBarbero4thGrade, ModelBarbero1stGrade y ModelBarberoSimplified . . . . .	20
3.2.2. Clase HCE para modelado del Heat Collector Element . . . . .	22
3.2.3. Clase SCA, para el modelado del Solar Collector Assembly . . . . .	28
3.2.4. Clase Loop . . . . .	30
3.2.5. Clase Subfield, modelado del subcampo . . . . .	31

3.2.6. Clase <b>Solarfield</b> , modelado del campo solar . . . . .	32
3.2.7. Clase <b>Fluid</b> y sus clases hijas, <b>FluidCoolProp</b> y <b>FluidTabular</b> . . . . .	33
3.2.8. Clases <b>Weather</b> , <b>FieldData</b> y <b>TableData</b> . . . . .	37
3.2.9. Clase <b>Site</b> . . . . .	38
3.3. Procedimiento para realizar una simulación . . . . .	38
3.4. Validación por comparación con otra herramienta de simulación . . . . .	41
3.4.1. Configuración de la simulación para comparación con SAM . . . . .	41
3.4.2. Resultados de la validación . . . . .	51
4. APLICACIONES DEL CÓDIGO DE SIMULACIÓN. . . . .	58
4.1. Aplicación para el análisis paramétrico . . . . .	58
4.1.1. Rendimiento del HCE en función de la radiación normal directa, <i>DNI</i> . . .	58
4.1.2. Rendimiento del HCE en función de la temperatura de entrada . . . . .	60
4.1.3. Rendimiento del HCE en función del flujo de radiación absorbido, $\dot{q}_{abs}''$ . .	62
4.1.4. Simulación con los diferentes modelos teóricos. . . . .	64
4.1.5. Simulación cambiando el tamaño de la malla de integración . . . . .	69
4.2. Análisis de los datos de generación de una planta solar termoeléctrica real .	75
5. CONCLUSIONES . . . . .	83
BIBLIOGRAFÍA . . . . .	84
GLOSARIO . . . . .	85
CÓDIGO FUENTE: CSENERGY.PY . . . . .	88
CÓDIGO FUENTE: INTERFACE.PY . . . . .	140



## ÍNDICE DE FIGURAS

1.1	Tecnologías solares de concentración . . . . .	3
1.2	Lazo de concentradores cilindroparabólicos . . . . .	4
1.3	Vista de perfil de un SCA . . . . .	5
2.1	Esquema del receptor empleado para el modelo . . . . .	9
3.1	Esquema relacional de las Clases HCE, SCA y Loop . . . . .	19
3.2	Esquema relacional de las Clases SolarField, Subfield y Loop . . . . .	20
3.3	Parámetros físicos del <i>Dowtherm-A</i> en estado de líquido saturado . . . . .	35
3.4	Configuración SAM. Selección del archivo de datos meteorológicos . .	43
3.5	Configuración SAM. Campo solar . . . . .	43
3.6	Configuración SAM. Configuración del SCA SenerTrough I a partir del modelo EuroTrough ET150 . . . . .	44
3.7	Configuración SAM. Configuración del HCE UVAC 3 con vacío . . . . .	45
3.8	Configuración del tipo de simulación . . . . .	47
3.9	Configuración de la simulación del campo solar . . . . .	48
3.10	Configuración de la simulación. Selección y configuración del fluido ca-	
	loportador . . . . .	49
3.11	Configuración de la simulación. Selección del modelo de SCA y configu-	
	ración . . . . .	50
3.12	Configuración de la simulación. Selección del modelo de HCE y configu-	
	ración . . . . .	50
3.13	Caudal másico calculado por SAM y por el código Python para el día 17/07/2007 . . . . .	52

3.14 Temperatura de entrada y de salida calculadas por SAM y por el código Python para el día 17/07/2007 . . . . .	53
3.15 Potencia térmica calculada por SAM y por el código Python para el día 17/07/2007 . . . . .	53
3.16 Caudal másico calculado por SAM y por el código Python para el día 17/06/2007 . . . . .	55
3.17 Temperatura de entrada y de salida calculadas por SAM y por el código Python para el día 17/06/2007 . . . . .	55
3.18 Potencia térmica calculada por SAM y por el código Python para el día 17/06/2007 . . . . .	56
4.1 Rendimiento térmico en función de DNI para diferentes modelos de HCE . . . . .	60
4.2 Rendimiento térmico en función de la temperatura de entrada del HTF para diferentes modelos de HCE . . . . .	62
4.3 Rendimiento térmico en función del flujo de radiación absorbido para diferentes modelos de HCE . . . . .	63
4.4 Temperaturas de salida obtenidas con los tres modelos . . . . .	66
4.5 Caudales de salida obtenidos con los tres modelos en un día de condiciones estables . . . . .	66
4.6 Potencia térmica obtenida con cada uno de los tres modelos en un día de condiciones estables . . . . .	67
4.7 Rendimiento calculado con cada modelo para un tamaño de malla de integración de 4,05 m . . . . .	70
4.8 Rendimiento calculado con cada modelo para un tamaño de malla de integración de 48,60 m . . . . .	71
4.9 Rendimiento calculado con cada modelo para un tamaño de malla de integración de 72,90 m . . . . .	72
4.10 Rendimiento calculado con cada modelo para un tamaño de malla de integración de 145,80 m . . . . .	72

4.11 Rendimiento calculado para diferentes tamaños de malla de integración . . . . .	73
4.12 Desviación, para simulaciones con el Modelo de 4º Orden, según diferentes tamaños de malla . . . . .	74
4.13 Desviación, para simulaciones con el Modelo de 1 <sup>er</sup> Orden, según diferentes tamaños de malla . . . . .	74
4.14 Desviación, para simulaciones con el Modelo de Simplificado, según diferentes tamaños de malla . . . . .	75
4.15 Vista aérea de las centrales Aste 1A y Aste 1B . . . . .	76
4.16 Reflectividad registrada durante el mantenimiento . . . . .	77
4.17 Temperaturas de operación reales y simuladas en un día de condiciones estables . . . . .	79
4.18 Potencia térmica real y simulada en un día de condiciones estables . . . . .	79
4.19 Caudal real y simulado en un día de condiciones estables . . . . .	80
4.20 Potencia térmica real y simulada en función de DNI . . . . .	81



## ÍNDICE DE TABLAS

3.1	Coeficientes polinómicos para el fluido caloportador <i>Dowtherm-A</i> . . . . .	36
3.2	Coeficientes polinómicos para el fluido caloportador <i>Therminol VP-1</i> . . . . .	37
3.3	Configuración del SCA modelo SenerTrough I de Sener . . . . .	44
3.4	Configuración del HCE modelo UVAC 3 de Solel . . . . .	46
3.5	Resultados globales anuales para las simulaciones con SAM y Python . . . . .	51
3.6	Resultados de las simulaciones en un día de condiciones estables . . . . .	54
3.7	Resultados de las simulaciones en un día de condiciones inestables . . . . .	57
4.1	Constantes del modelo de emisividad equivalente para cada uno de los receptores seleccionados . . . . .	58
4.2	Rendimiento en función de la radiación normal incidente para distintos modelos de HCE . . . . .	59
4.3	Rendimiento en función de la temperatura de entrada del HTF para diferentes modelos de HCE . . . . .	61
4.4	Rendimiento en función del flujo de radiación absorbido y para cada modelo teórico . . . . .	64
4.5	Temperaturas obtenidas en la simulación con cada modelo teórico en un día de condiciones estables . . . . .	65
4.6	Caudales obtenidos en la simulación para cada modelo en un día de condiciones estables . . . . .	68
4.7	Potencia térmica calculada en la simulación de cada modelo en un día de condiciones estables . . . . .	69



# 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 y sus diferentes librerías para el cálculo científico (Numpy) y el tratamiento de datos (Pandas). Finalmente, todo el código se encuentra publicado y accesible a través de GitHub y se puede interactuar con la versión final del código de simulación a través de un Notebook Jupyter.

Los resultados obtenidos son compatibles con los datos de generación de un campo solar real dentro de las limitaciones que los diferentes modos de operación y eventos impredecibles (paradas de planta para mantenimiento, disparos por avería, etc...) introducen en el proceso. En este proyecto solo se aborda la simulación del sistema del campo solar, el único sistema dentro del alcance del modelo teórico de partida. Para la simulación de planta sería necesario el desarrollo de modelos para gran número de sistemas como por ejemplo, los generadores de vapor, almacenamiento térmico, turbina, sistemas auxiliares de bombeo, sistemas de tratamiento de agua, torre de refrigeración, etc... No obstante, 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 validación 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 central de torre y la de CCP son las que cuentan en la actualidad con una mayor madurez y gran número de centrales en operación y en construcción en todo el mundo [2]. En España, actualmente hay medio centenar de centrales CCP en operación [3], 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 [3]

(b) Concentradores de Disco Parabólico. Fuente [3]



(c) Colector Cilindro-Parabólico. Fuente: Propia

(d) Concentrador de tipo Fresnel. Fuente [3]

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 energía luminosa en energía térmica con una buena eficiencia y que puede alcanzar temperaturas de hasta 675 K.



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: Propia

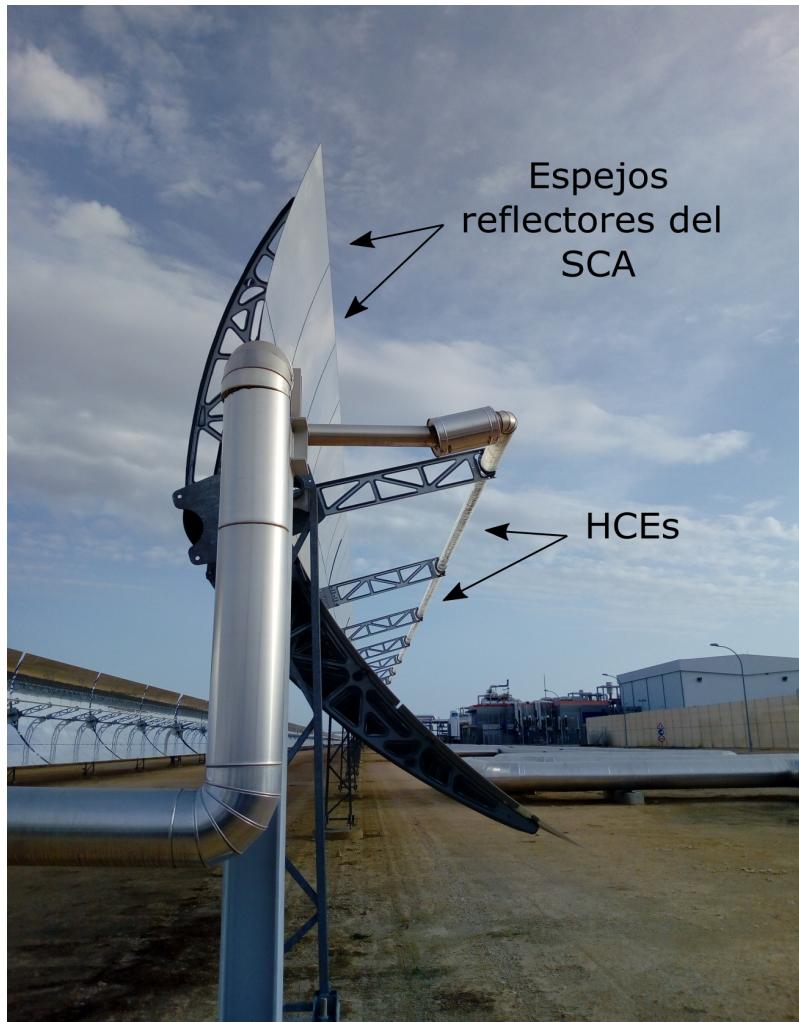


Fig. 1.3. Vista de perfil de un SCA. Pueden distinguirse algunos HCEs y los brazos de soporte. Fuente: Propia

### 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**

El rango de temperatura ideal para trabajar con colectores cilindro parabólicos es de 425 K a 675 K. Para temperaturas superiores las pérdidas térmicas son altas y para temperaturas inferiores hay otros colectores más económicos (colectores de placa plana y colectores de tubo de vacío). El tipo de fluido a emplear depende de la temperatura de trabajo. 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. Actualmente se emplean también sales fundidas y están en desarrollo sistemas de generación directa de vapor (emplean directamente el agua como fluido caloportador).

## **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 [4], [5], [6] o [7]) y, por otro lado, modelos teóricos más generales como los de [8], [9] o [10]. 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 [11]. 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 consistente 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,  $\varepsilon_{ext}$  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.

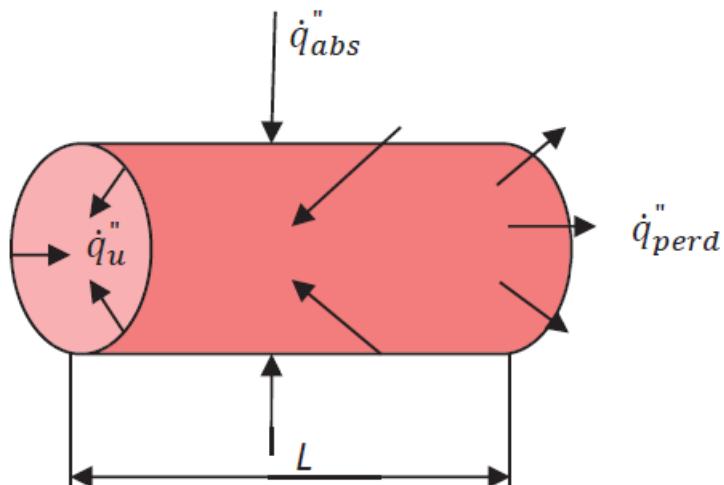


Fig. 2.1. Esquema del receptor empleado para el modelo. Fuente:[1]

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  $\eta_{opt}$ , las pérdidas geométricas  $\eta_{bordes}$  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):

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

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<sup>er</sup> grado.

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

donde  $\sigma$  es la constante de Stefan-Boltzmann ( $5.67 \times 10^{-8} \text{ W/m}^2\text{K}^4$ ),  $\varepsilon_{ext}$  es la emisividad del tratamiento superficial exterior y  $h_{hext}$  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  $T_{fe}$  a la temperatura del fluido a la entrada, tenemos:

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

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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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  $\varepsilon_{ext}$  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:

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

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 [6], [7] y [12].

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,  $\eta_0$ , por su valor aproximado dado en la ec.(2.26):

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

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

En esta ecuación se han reagrupado variables en diferentes términos que cuentan con sentido físico. De este modo, se definen:

$$\dot{q}_{crit}'' = \sigma \cdot \varepsilon_{ext} \cdot (T_{fe}^4 - T_{ext}^4) + h_{hext} \cdot (T_{fe}^4 - T_{ext}^4) \quad (2.27)$$

y

$$U_{crit} = 4 \cdot \sigma \cdot \varepsilon_{ext} \cdot T_{fe}^3 + h_{hext} \quad (2.28)$$

Los coeficientes  $\dot{q}_{crit}''$  y  $U_{crit}$  son valores de referencia en el estudio del comportamiento del colector, pues cuando  $\dot{q}_{abs}''$  y  $U_{rec}$  se aproximan a ellos el rendimiento del colector se hace nulo. Por otra parte,  $F'_{crit}$  se asemeja al parámetro empleado en el modelo desarrollado por Hottel y Whillier en [8].

$$F'_{crit} = \frac{1}{\frac{4 \cdot \sigma \cdot \varepsilon_{ext} \cdot T_{fe}^3}{U_{rec}} + \frac{h_{hext}}{U_{rec}} + 1} = \frac{1}{\frac{U_{crit}}{U_{rec}} + 1} \quad (2.29)$$

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,  $\eta_0$ .

### 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.(2.30) para el cálculo del rendimiento para la totalidad del receptor,  $\eta_T$ , mediante el Modelo Simplificado:

$$\begin{aligned}\eta_T &= F'_{crit} \cdot \left[ 1 - \frac{\dot{q}''_{crit}}{\dot{q}''_{abs}} \right] \\ &= \frac{F'_{crit}}{Cg} \cdot \left[ Cg \cdot IAM \cdot \cos(\theta) \cdot \eta_{opt,pico} \cdot \eta_{sombras} \cdot \eta_{bordes} - \frac{h_{ext} \cdot (\bar{T}_f - T_{ext})}{DNI} \right. \\ &\quad \left. - \frac{\sigma \cdot \varepsilon_{ext} \cdot (\bar{T}_f^4 - T_{ext}^4)}{DNI} \right]\end{aligned}\quad (2.30)$$

Esta ecuación es más parecida a la encontrada en otros modelos de diferentes autores, como por ejemplo en [8] o [9], 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  $U_{rec}$ ,  $\varepsilon_{ext}$ ,  $h_{ext}$  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 trasmisión 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.
- Dada la longitud real del tubo absorbedor en una planta CCP, puede 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 mayores restricciones, especialmente en los rangos de operación aceptables (no deben ser próximos a los valores críticos definidos en la ec.(2.28) y (2.27)), siendo sus resultados algo menos precisos en general.

El Modelo Simplificado solo es válido además para longitudes de receptor 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 SCE 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.

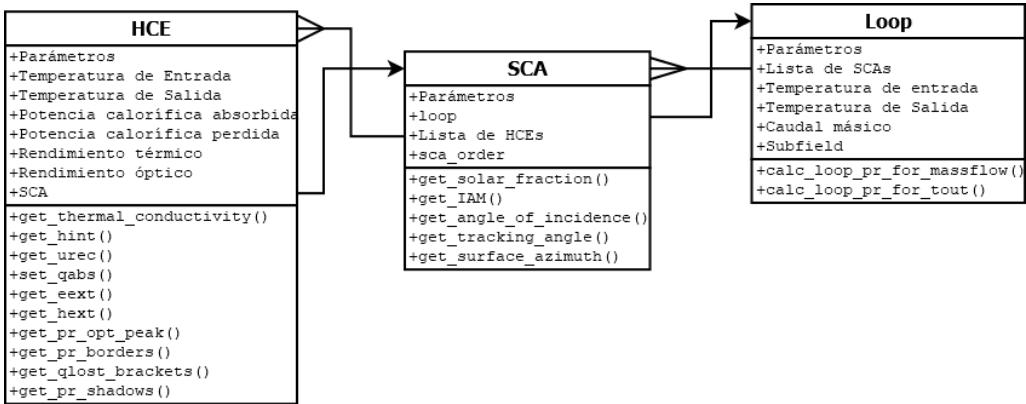


Fig. 3.1. Esquema relacional de las Clases HCE, SCA y Loop. Se muestran los principales atributos y métodos de cada clase

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.

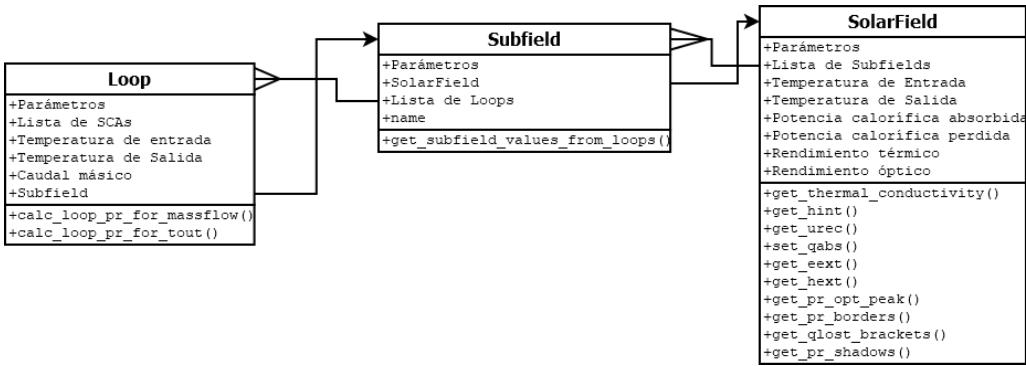


Fig. 3.2. Esquema relacional de las Clases SolarField, Subfield y Loop. Se muestran los principales atributos y métodos de cada clase

### 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 un 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  $q_{abs}$ .

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 adsorbido 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 (2.27) y (2.28) 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  $\eta_0$ . Como valor inicial se calcula  $\eta_0$  a partir de la ec.(2.26) del Modelo de 1<sub>er</sub> Orden.
- Con el valor de  $\eta_0$  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.

Para el primer HCE del lazo se emplea como temperatura del fluido  $T_f$  la temperatura del HTF a la entrada del HCE. Para los siguientes HCEs del lazo se incrementa la

temperatura de entrada con la mitad del salto de temperatura que experimentó el HCE anterior.

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 [13] para análisis unidimensional.

### **Clases ModelBarbero1stOrder y ModelBarberoSimplified**

El cálculo del rendimiento que realizan estas dos clases es idéntico al del Modelo de 4º Orden hasta el momento de llegar al proceso iterativo, punto en el cual se calcula el rendimiento térmico directamente mediante las ecuaciones (2.25) y (2.30) respectivamente.

#### **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: la 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  $C$ ); la potencia calorífica absorbida a lo largo de todo el HCE,  $q_{abs}$  (W); la potencia calorífica perdida a lo largo de todo el HCE,  $q_{lost}$  (W); el rendimiento óptico del conjunto HCE+SCA,  $\eta_{opt}$ ; el rendimiento térmico,  $\eta$  y la temperatura de salida,

$T_{out}$ , (K).

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

Con estos par\'ametros el comportamiento del HCE queda totalmente caracterizado en el sistema desde el punto de vista del proceso de generaci\'on. Estos atributos (pueden entenderse como variables) est\'an relacionados entre s\'i seg\'un las reglas que aplique cada modelo. La temperatura de salida del HTF,  $T_{out}$ , aparece impl\'icita en la ec.3.2:

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

donde  $\Delta H$  es el incremento de entalp\'ia del fluido, pues hemos considerado que se trata de un fluido incompresible y tambi\'en se ha despreciado la participaci\'on de energ\'ia cin\'etica. Previamente se debe calcular  $\Delta H$  seg\'un la ec.(3.3):

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

La ec.(3.2) puede resolverse por m\'etodos num\'ericos si el calor espec\'ifico  $C_p$  del fluido se ha obtenido a partir de un polinomio. En el caso de que se disponga de una funci\'on que proporcione la temperatura del fluido en funci\'on de la entalp\'ia  $T(H)$ , como ocurre si se usa *CoolProp*, se puede calcular su valor directamente a trav\'es de las funciones que ofrece esta librer\'ia como  $T_{out} = T(H_{out})$ .

En la ec.3.3 se introducen el *factor de longitud efectiva*,  $\gamma_L$  y *factor de interceptaci\'on geom\'etrico*,  $\gamma_g$  para tener en cuenta la reducci\'on de la longitud *activa* del HCE debido a los fuelles en los extremos del HCE y al sombreado del escudo t\'ermico en las uniones de HCEs. Un valor t\'ipico para ambos factores est\'a comprendido entre 0,96 y 0,97 [14]. En el caso de que el calor absorbido sea nulo, la temperatura de salida ser\'a inferior a la de entrada y el valor  $\Delta H < 0$ . En este caso, no existe reducci\'on de la longitud efectiva del absorbador y la energ\'ia perdida se calcula seg\'un la ec.(3.4) pues a lo largo de toda la superficie del HCE se experimentan p\'erdidas energ\'eticas:

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

## 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,  $A_r$ . 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:

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

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  $\theta$  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  $\rho$  (reflectividad del concentrador) y  $\gamma$  (fracción solar), son parámetros del SCA y deben obtenerse de la instancia de SCA asociada al HCE.  $\alpha$  es la absorvidad del receptor y  $\tau$  es la transmisiad 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  $\eta_{bordes}$  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  $\theta$ :

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

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 separación 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 [15] 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, [16]:

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

donde *beta* es el ángulo de seguimiento,  $D_L$  es la distancia de separación entre lazos y  $A_p$  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:

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

donde  $\varepsilon_{ext}$  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 4 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 [13]:

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

donde  $P_b$  es el perímetro de la sección del soporte,  $A_{cs,b}$  es la sección transversal de la unión entre el brazo y el tubo absorbador,  $K_b$  es la conductividad térmica del acero empleado en el brazo,  $\bar{h}_b$  es el coeficiente de transmisión de calor por convección medio hacia el exterior,  $T_{base}$  es la temperatura de la zona de conexión entre los brazos y el tubo absorbador,  $L$  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 absorbador 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 absorvedad solar  $\alpha_{solar}$ , la transmisividad del vidrio  $\tau$  y su reflectividad  $\rho$ .

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  $\beta$  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 indice realmente sobre el tubo absorbedor. Su valor se obtiene según la ec.(3.18) como producto de una serie de factores:

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

El factor geométrico  $\eta_{geometrico}$  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  $\beta$  en el plano de apertura del reflector. Las expresiones generales pueden encontrarse en [11] pero, en nuestro caso, hemos recurrido a la librería *pvlb-python* [17], 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 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):

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

Para el caso del *Dowtherm-A* se han obtenido los coeficientes de los polinomios que caracterizan cada parámetro a partir de [18] y se han contrastado las curvas con los datos ofrecidos en [19]. 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*.

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

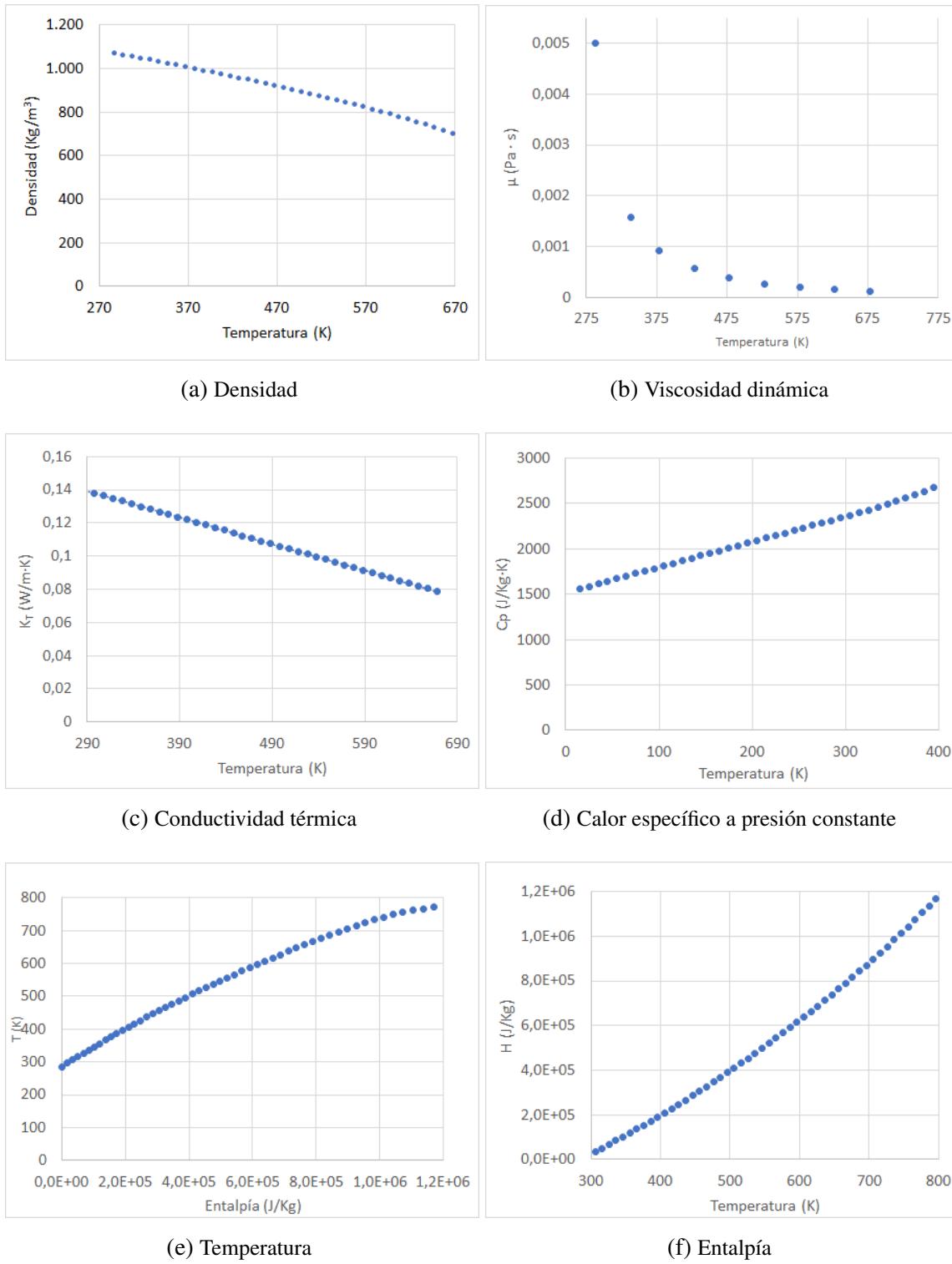


Fig. 3.3. Parámetros físicos del *Dowtherm-A* en estado de líquido saturado. Algunos puntos se han extrapolado más allá de la temperatura máxima de trabajo con el fin de evitar errores durante los procesos de convergencia.

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

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

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:

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

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

### 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, caudales, 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` que nos permite obtener información relativa a la posición solar para cada fecha del año.

## 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`.

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*.
- Se crea una instancia para el modelado del HTF a partir de la Clase `FluidCoolProp` si los datos se van a tomar desde la librería externa `CoolProp` o de la Clase `FluidTabular` si se le pasan los factores de los polinomios que permiten calcular cada parámetro del fluido.
- 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 `textttrunSimulation`.

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 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.

### **3.4. Validación por comparación con otra herramienta de simulación**

Con el fin realizar una primera validación 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, [20]), 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.3.4 se puede ver la pestaña de configuración de SAM para seleccionar el origen de datos meteorológicos.

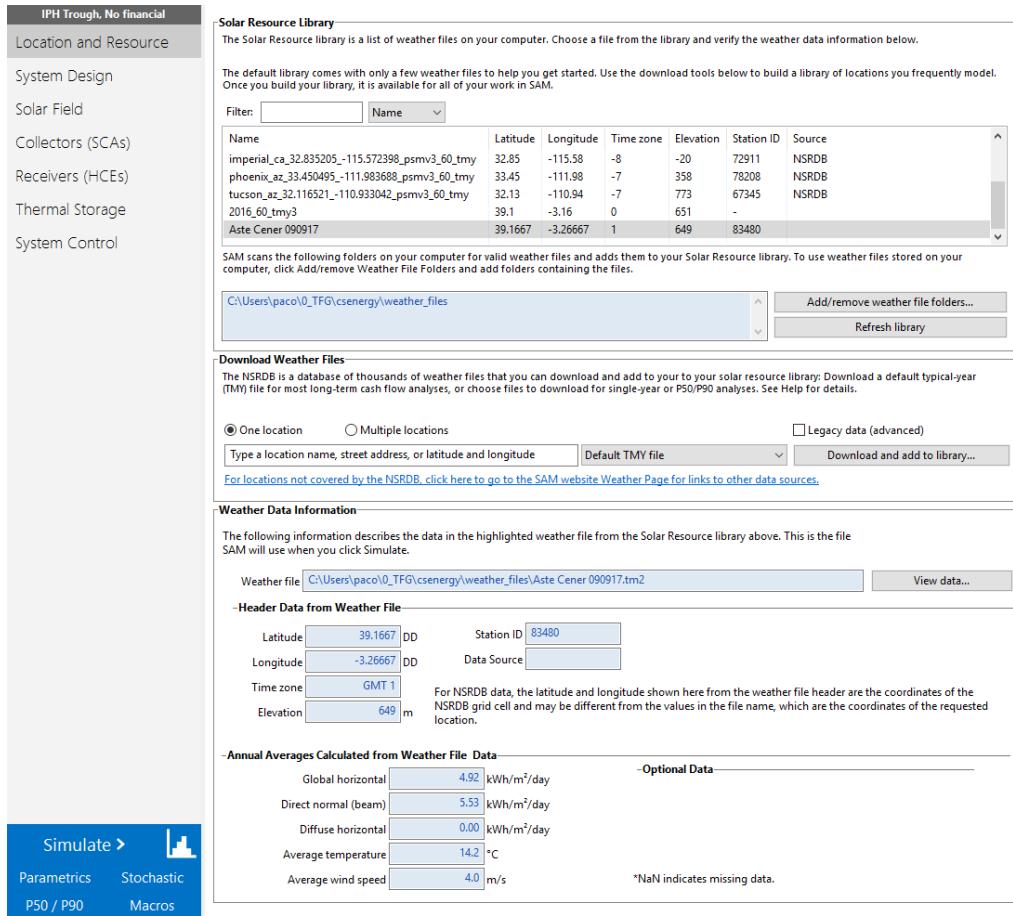


Fig. 3.4. Configuración SAM. Selección del archivo de datos meteorológicos

La versión de SAM utilizada ha sido la "2020.2.29, 64 bit, updated to revision 1". En la Fig.3.5 podemos ver la pestaña de configuración del campo solar.

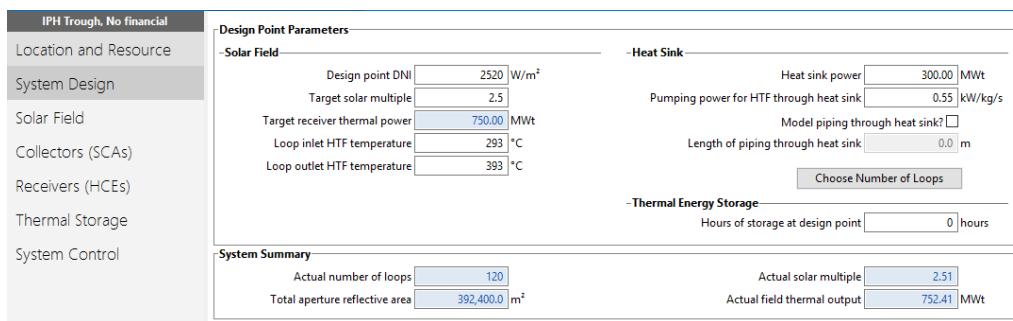


Fig. 3.5. Configuración SAM. Campo solar

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

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

TABLA 3.3. Configuración del SCA modelo SenerTrough I de Sener

En la Fig.3.6 puede verse una captura de pantalla de la pestaña de configuración del tipo de SCA en SAM. En la Fig.3.7, 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 3.4.

La captura de pantalla muestra la configuración del SCA SenerTrough I en el software SAM. El menú lateral izquierdo incluye: IPH Trough, No financial, Location and Resource, System Design, Solar Field, Collectors (SCAs), Receivers (HCEs), Thermal Storage y System Control. La configuración se divide en tres secciones principales:

- Design Point Parameters - Solar Field:** Muestra los siguientes valores:
  - 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:** Muestra los siguientes valores:
  - Heat sink power: 300,00 MWt
  - Pumping power for HTF through heat sink: 0,55 kW/kg/s
  - Model piping through heat sink? (checkbox)
  - Length of piping through heat sink: 0,0 m
  - Choose Number of Loops (botón)
- Thermal Energy Storage:** Muestra el valor de Hours of storage at design point: 0 hours.

**System Summary:** Muestra los siguientes valores:
 

- Actual number of loops: 120
- Actual solar multiple: 2,51
- Total aperture reflective area: 392,400,00 m²
- Actual field thermal output: 752,41 MWt

Fig. 3.6. Configuración SAM. Configuración del SCA SenerTrough I a partir del modelo EuroTrough ET150

**IPH Trough, No financial**

**Solar Field Design Point**

Single loop aperture	3,270 m <sup>2</sup>
Loop optical efficiency	0.772
Total loop conversion efficiency	0.761
Total required aperture, SM=1	156,456 m <sup>2</sup>
Required number of loops, SM=1	48
Total tracking power	60,000 W
Actual number of loops	120
Total aperture reflective area	392,400 m <sup>2</sup>
Actual solar multiple	2.51
Actual field thermal output	752,415 MWt
Loop inlet HTF temperature	293 °C
Loop outlet HTF temperature	393 °C

**Solar Field Parameters**

Row spacing	16.25 m
Header pipe roughness	4.57e-05 m
HTF pump efficiency	0.85
Piping thermal loss coefficient	0 W/m <sup>2</sup> -K
Wind stow speed	25 m/s
Receiver startup delay time	0 hr
Receiver startup delay energy fraction	0 -
Collector startup energy	0 kWh/eca
Tracking power per SCA	125 W/eca
Number of field subsections	1
Allow partial defocusing	Simultaneous <input checked="" type="checkbox"/>

**Heat Transfer Fluid**

Field HTF fluid	Therminol VP-1
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	2 m/s
Hot Headers	2 m/s
Header design min flow velocity	2 m/s
Header design max flow velocity	10 m/s

**Collector Orientation**

Collector tilt	0 deg
Tilt: horizontal=0, vertical=90	
Collector azimuth	0 deg
Azimuth: equator=0, west=90	
Stow angle	170 deg
Deploy angle	10 deg

**Mirror Washing**

Water usage per wash	0.7 L/m <sup>2</sup> , aper.
Washes per year	12

**Plant Heat Capacity**

Hot piping thermal inertia	0 kWht/K-MWt
Cold piping thermal inertia	0 kWht/K-MWt
Field loop piping thermal inertia	0 Wht/K-m

**Land Area**

Solar field area	273 acres
Non-solar field land area multiplier	1.1
Total land area	300 acres

**Single Loop Configuration**

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.

Number of SCA/HCE assemblies per loop:   Edit SCAs  Edit HCEs  Edit Defocus Order

SCA: 1	SCA: 1	SCA: 1	SCA: 1
HCE: 1 DF# 4	HCE: 1 DF# 3	HCE: 1 DF# 2	HCE: 1 DF# 1

**Simulate >** 

Parametrics    Stochastic  
P50 / P90    Macros

Fig. 3.7. Configuración SAM. Configuración del HCE UVAC 3 con vacío

Parámetro	Valor
Logintud	4,05 (m)
Factor de longitud efectiva, $\gamma_L$	0,96
Factor de interceptación geométrica, $\gamma_g$	0,96
Diámetro interior del tubo absorbedor, $d_{ri}$	0,066 (m)
Diámetro exterior del tubo absorbedor, $d_{ror}$	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)

TABLA 3.4. Configuración del HCE modelo UVAC 3 de Solel

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

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 que sirve de guía durante el proceso. En la Fig.3.8 se muestra la primera pantalla de la interfaz de configuración. El archivo de origen de datos incluye los mismos datos meteorológicos 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.

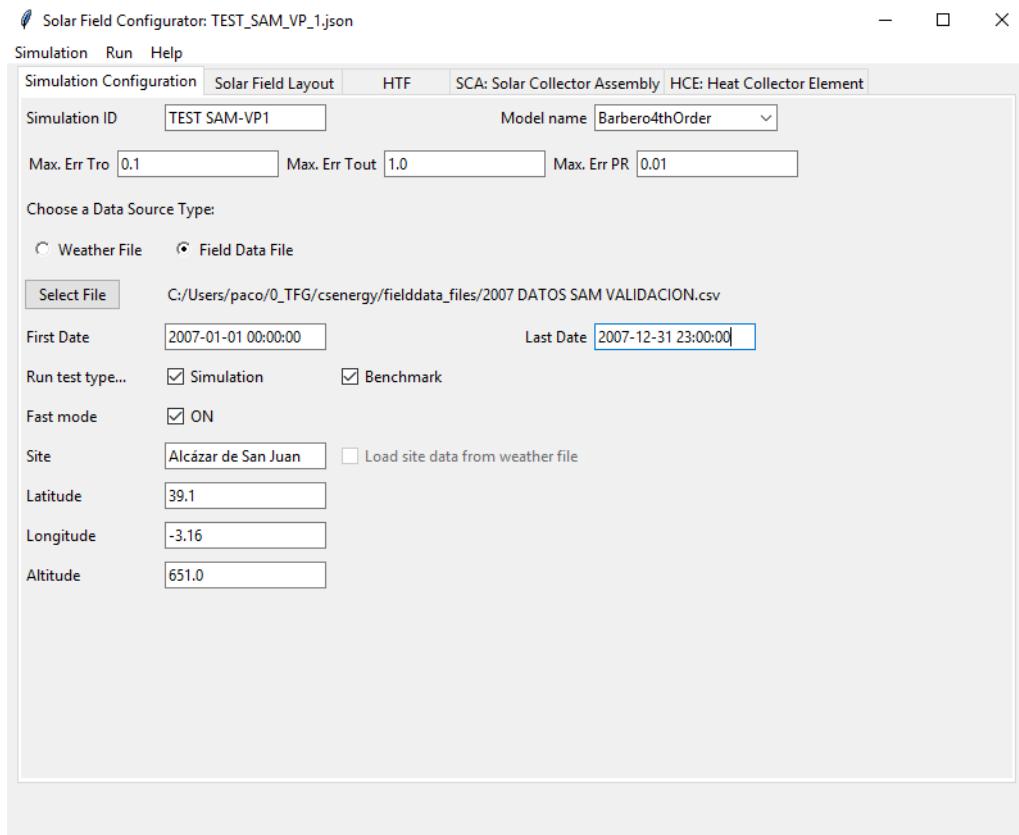


Fig. 3.8. Configuración de la simulación. Selección del tipo de simulación, fichero de datos y lugar de emplazamiento.

La configuración del campo solar es similar a la de SAM. 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 393C (666,15 K).

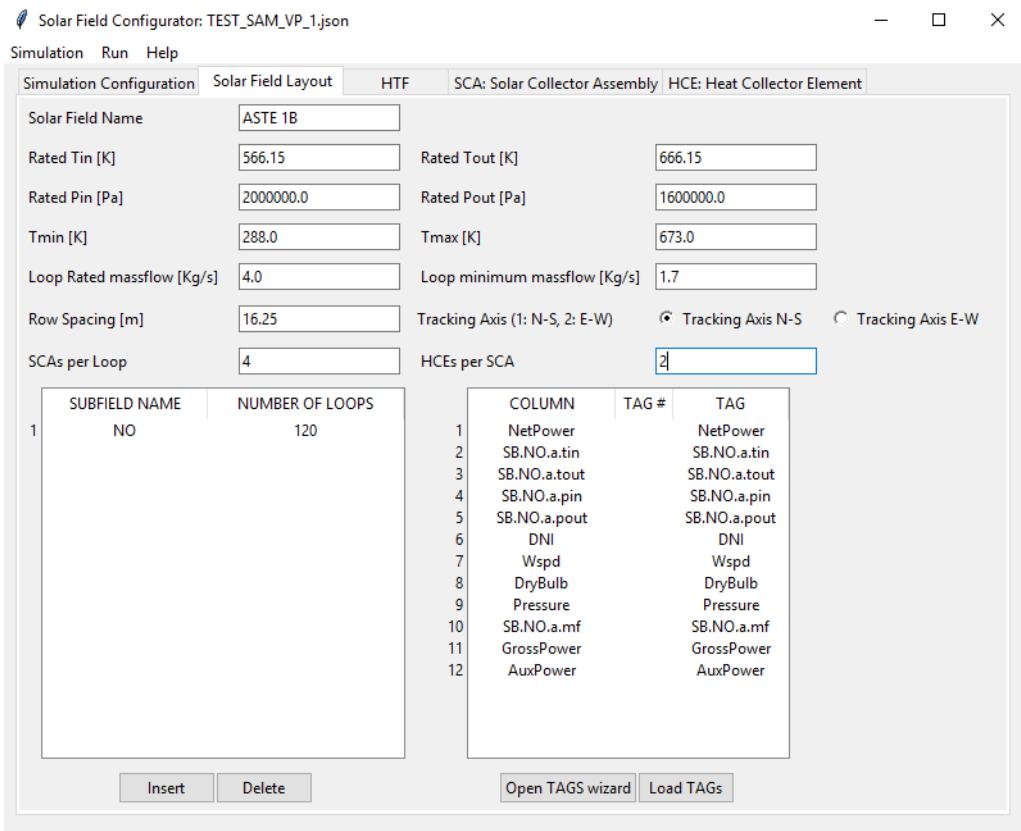


Fig. 3.9. Configuración de la simulación. Configuración del campo solar, número de subcampos, lazos, configuración de los lazos, valores nominales y asistente para relacionar los identificadores de las columnas del archivo de origen de datos con los que maneja el programa.

El fluido caloportador *Therminol VP-1* se configura tal y como puede verse en la Fig.3.10. Los coeficientes se cargan desde una librería donde los hemos guardado previamente tras su obtención según se indicó en el apartado 3.2.7.

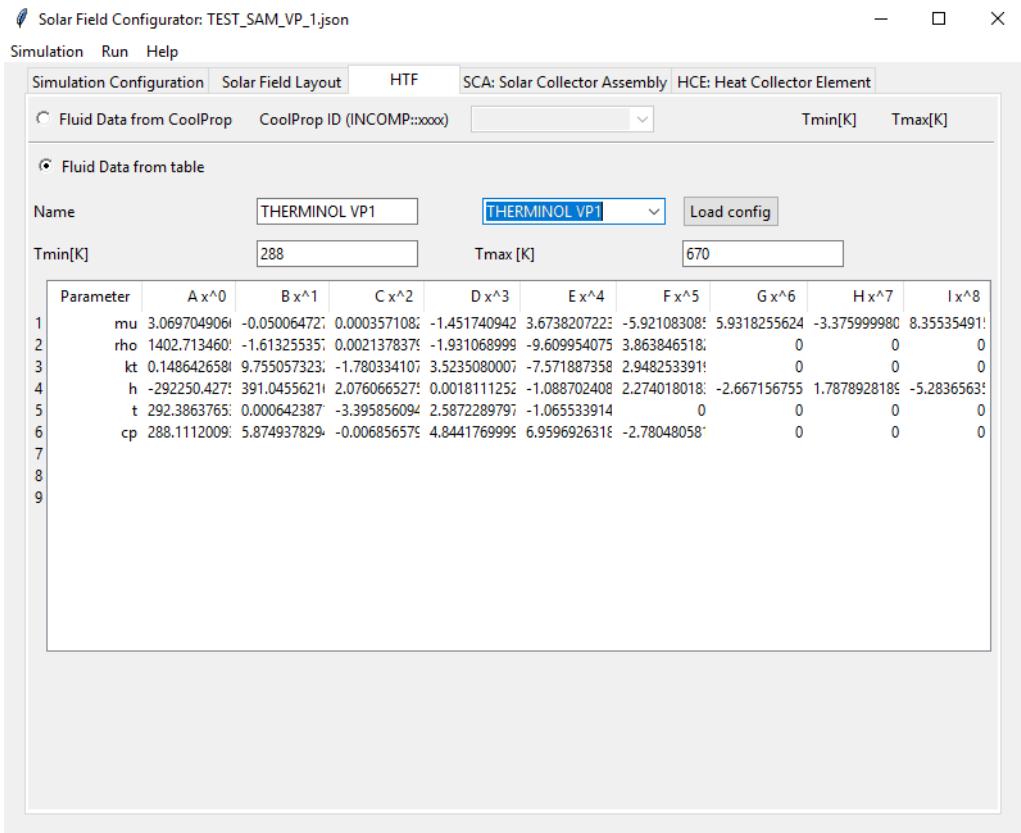


Fig. 3.10. Configuración de la simulación. Selección y configuración del fluido calorportador

Finalmente, en las figuras 3.11 y 3.12 puede verse la configuración del tipo de SCA y HCE respectivamente.

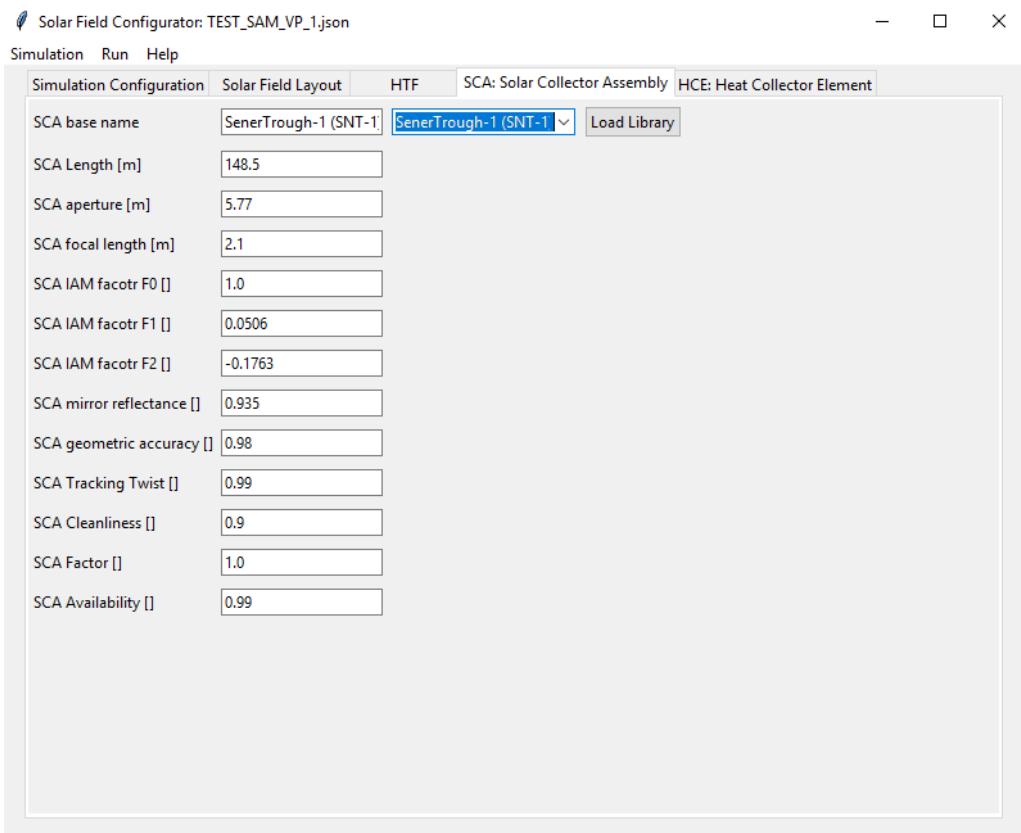


Fig. 3.11. Configuración de la simulación. Selección del modelo de SCA y configuración

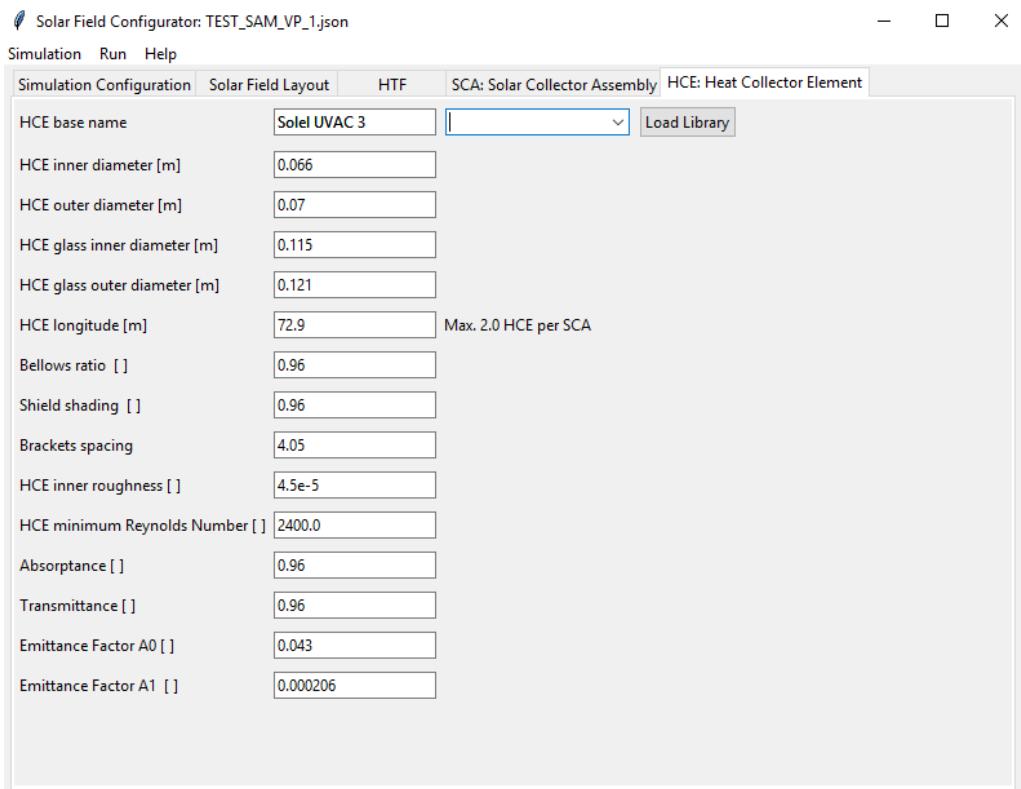


Fig. 3.12. Configuración de la simulación. Selección del modelo de HCE y configuración

### 3.4.2. Resultados de la validación

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,  $\dot{q}_{abs}''$ .

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

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

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 ambas 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 3.13 a 3.15 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 3.16 a 3.18 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.

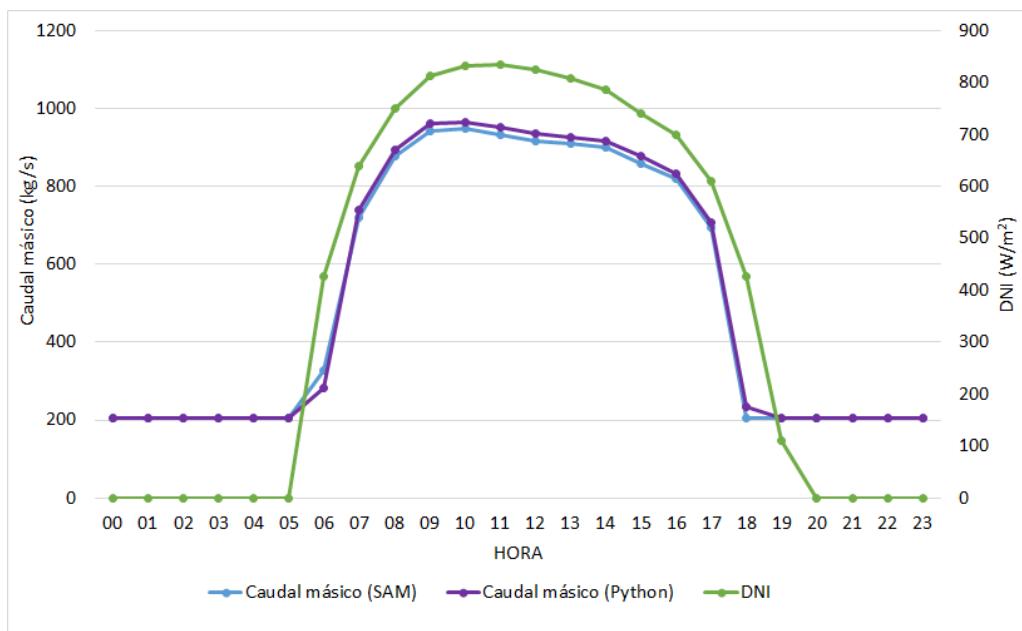


Fig. 3.13. Caudal másico calculado por SAM y por el código Python para el día 17/07/2007

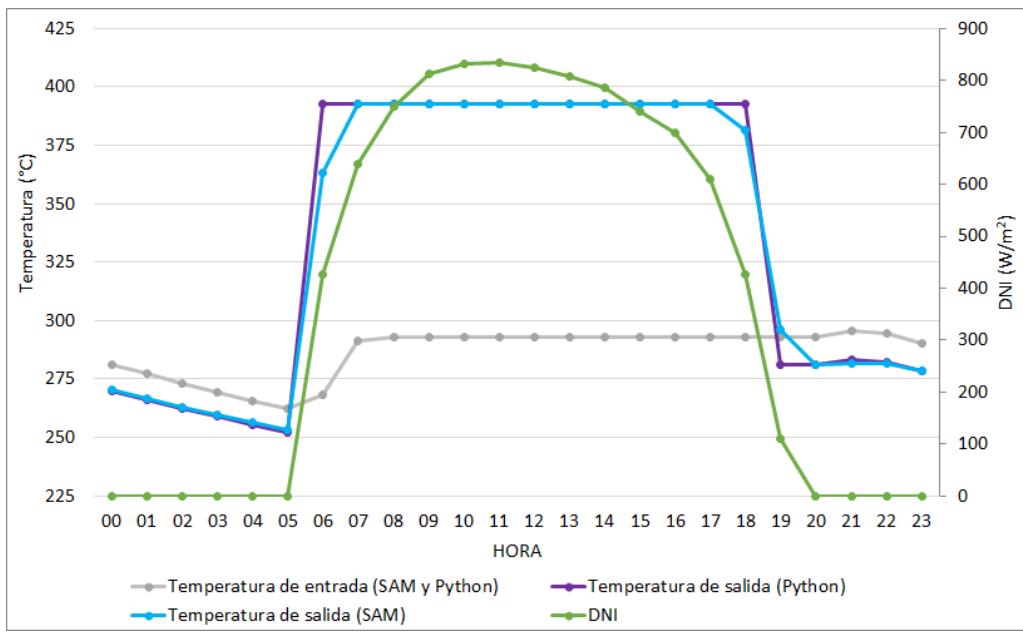


Fig. 3.14. Temperatura de entrada y de salida calculadas por SAM y por el código Python para el día 17/07/2007

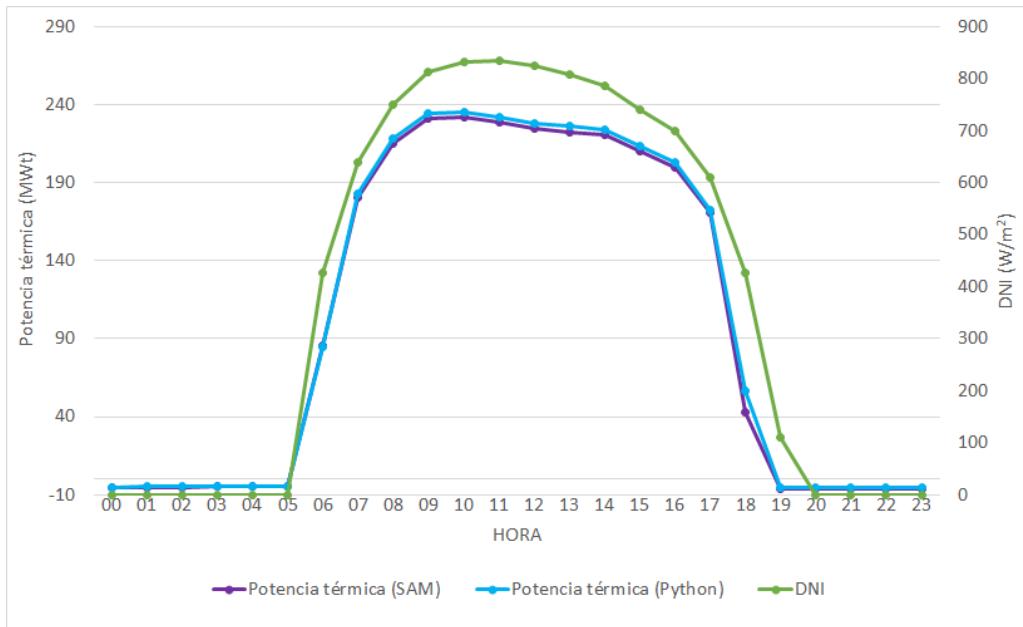


Fig. 3.15. Potencia térmica calculada por SAM y por el código Python para el día 17/07/2007

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

Hora	DNI (S, P) (W/m <sup>2</sup> )	Caudal (S) (Kg/s)	Caudal (P) (Kg/s)	T <sub>in</sub> (S, P) (°C)	T <sub>out</sub> (S) (°C)	T <sub>out</sub> (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

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

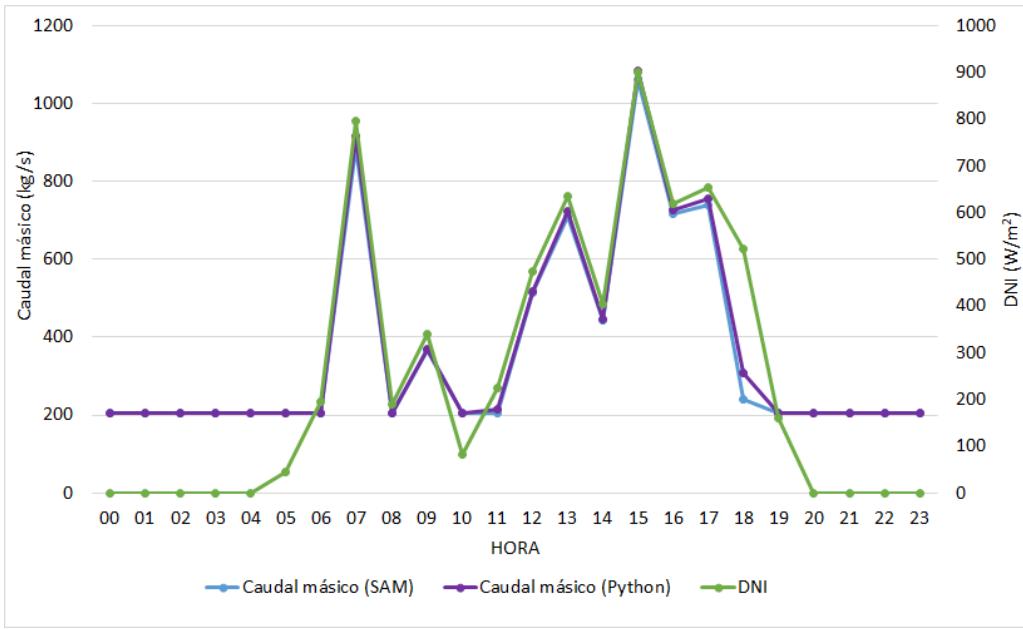


Fig. 3.16. Caudal másico calculado por SAM y por el código Python para el día 17/06/2007

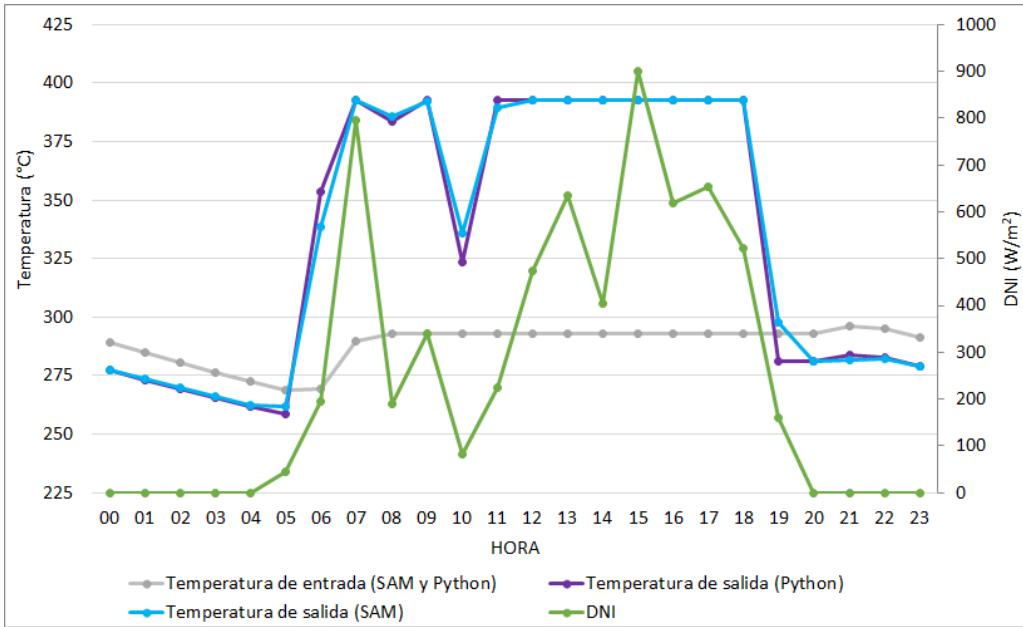


Fig. 3.17. Temperatura de entrada y de salida calculadas por SAM y por el código Python para el día 17/06/2007

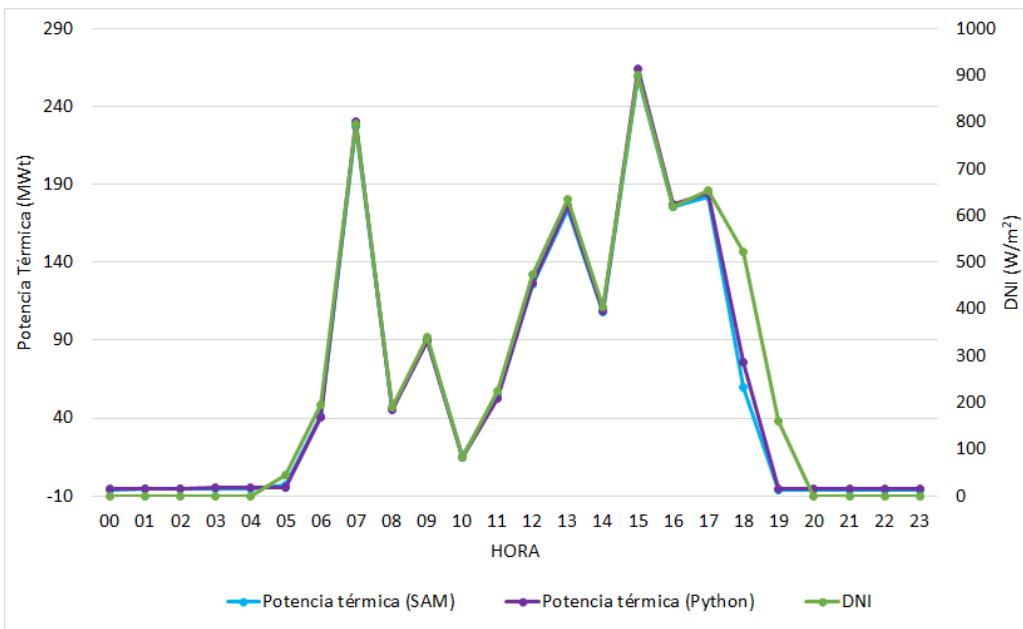


Fig. 3.18. Potencia térmica calculada por SAM y por el código Python para el día 17/06/2007

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

Hora	DNI (S, P) (W/m <sup>2</sup> )	Caudal (S) (Kg/s)	Caudal (P) (Kg/s)	T <sub>in</sub> (S, P) (°C)	T <sub>out</sub> (S) (°C)	T <sub>out</sub> (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
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

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

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  $\varepsilon_{ext}$  conforme a la ec.(3.12) y el valor de la absorptividad de algunos modelos de HCE según se recogen en [1].

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*
ASE HEMS08	2,03E-04	-1,03E-02	95
NREL #6	1,52E-04	1,96E-03	96

TABLA 4.1. Constantes del modelo de emisividad equivalente para cada uno de los receptores seleccionados. \*Al no disponerse de este dato, se ha empleado este valor con el fin de poder incluir el modelo en el test.

Fuente [1]

Si simulamos el comportamiento de un HCE de cada modelo para diferentes valores de DNI, bajo las mismas condiciones de caudal y temperatura de entrada del HTF, po-

demos 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 validación con SAM. Para la fecha y hora de simulación se empleará el 1 de julio a las 12:00 UTC (14:00 hora local).

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.

$DNI(W/m^2)$	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

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

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.

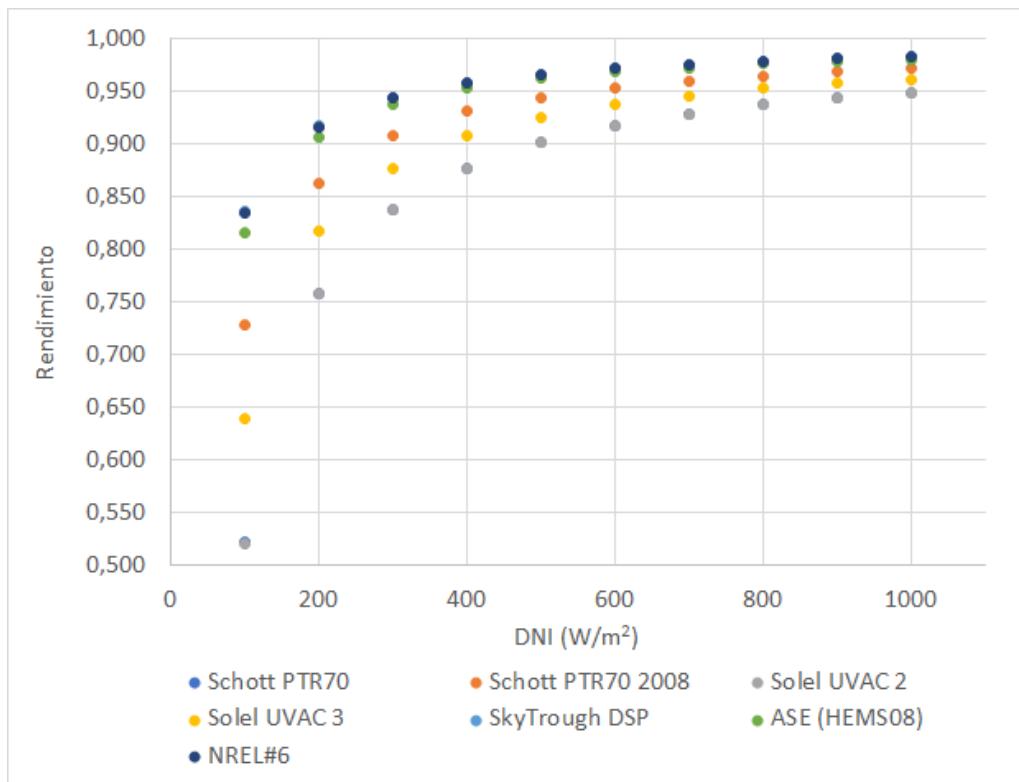


Fig. 4.1. Rendimiento térmico en función de DNI para diferentes modelos de HCE.

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

#### 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.

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

$T_{in}(^{\circ}C)$	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
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

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

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.

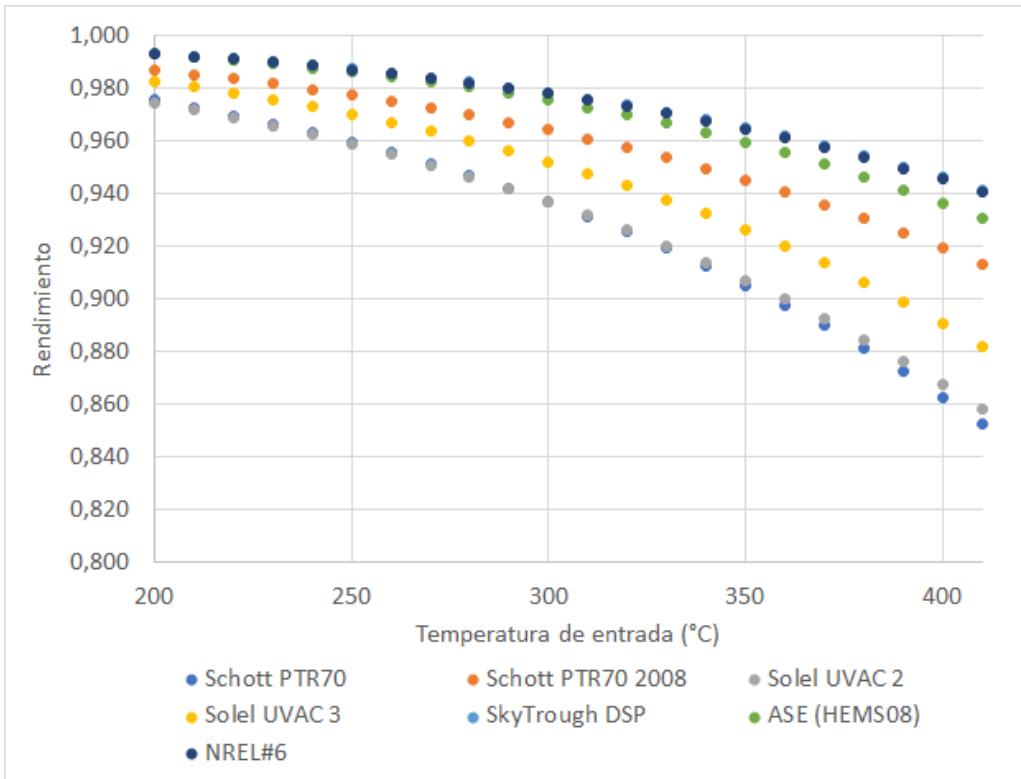


Fig. 4.2. Rendimiento térmico en función de la temperatura de entrada del HTF para diferentes modelos de HCE.  $DNI = 800W/m^2$ ,  $\dot{m} = 6 \text{ kg/s}$

#### 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.

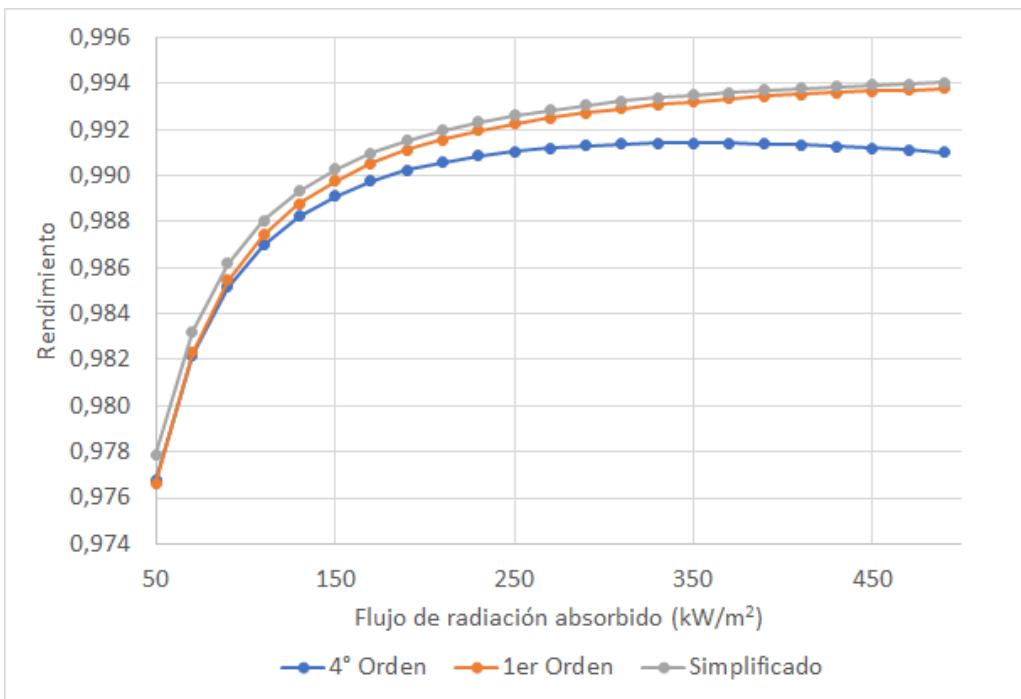


Fig. 4.3. Rendimiento térmico en función del flujo de radiación absorbido para diferentes modelos de HCE.  $t_{in} = 300 \text{ }^{\circ}\text{C}$ ,  $\dot{m} = 6 \text{ kg/s}$

Los valores correspondientes se muestran en la tabla 4.4.

$\dot{q}_{abs}''$	4° Orden	1 <sup>er</sup> 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
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

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

#### 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<sup>er</sup> 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).

Hora	DNI (W/m <sup>2</sup> )	$T_{in}$ (°C)	$T_{out}$ (SAM) (°C)	$T_{out}$ (4 <sup>o</sup> Ord.) (°C)	$T_{out}$ (1 <sup>er</sup> 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
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

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

El resultado gráfico de este test puede versen 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.

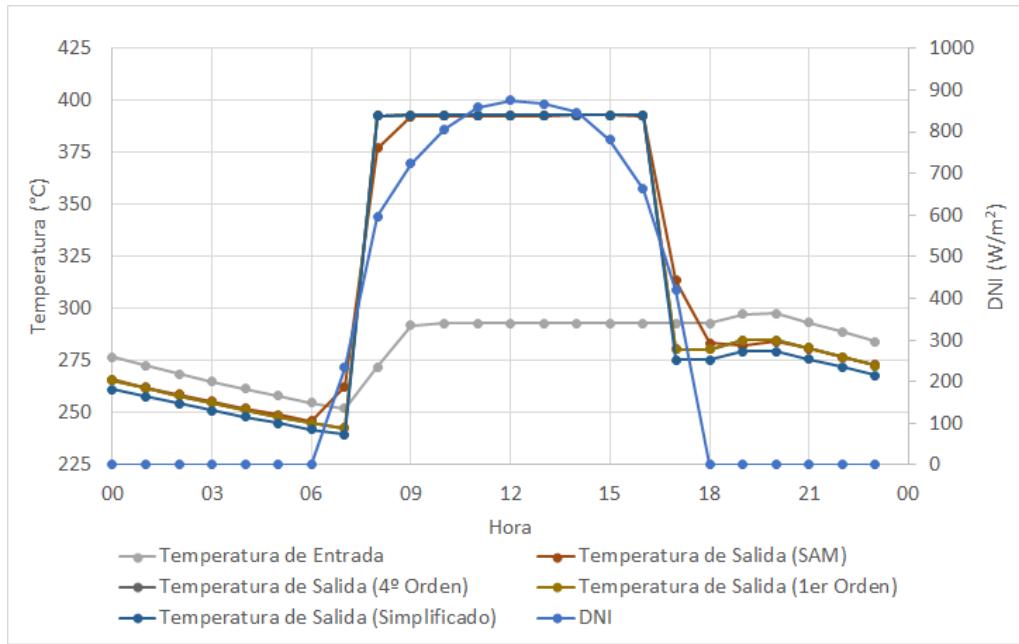


Fig. 4.4. Temperaturas de salida obtenidas con los tres modelos. Simulación con los datos del día 2/3/2007.

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.

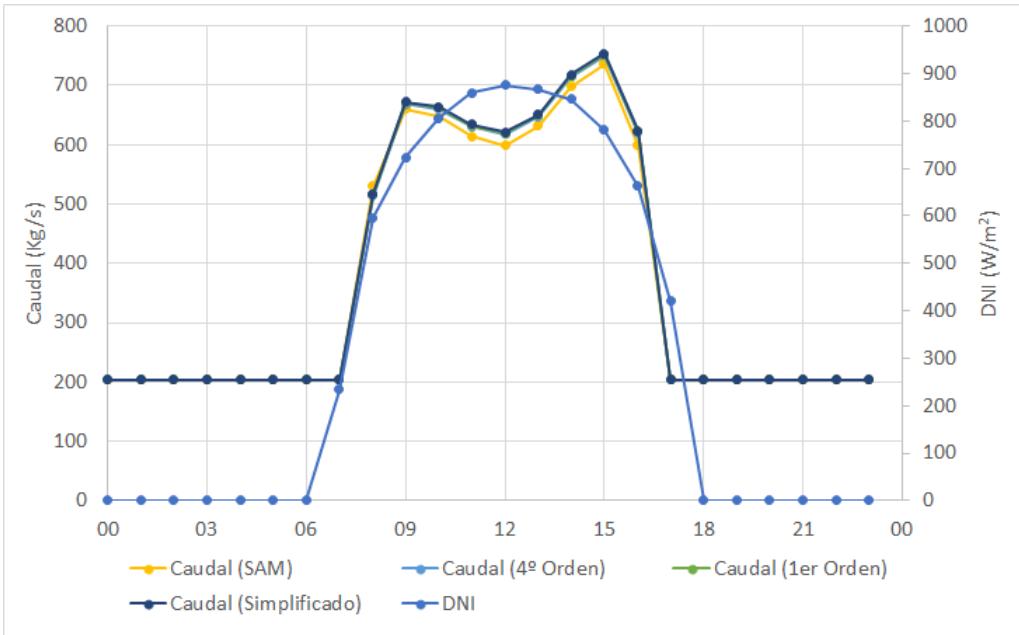


Fig. 4.5. Caudales de salida obtenidos con los tres modelos. Datos del día 2/3/2007.

Condiciones estables y buena radiación

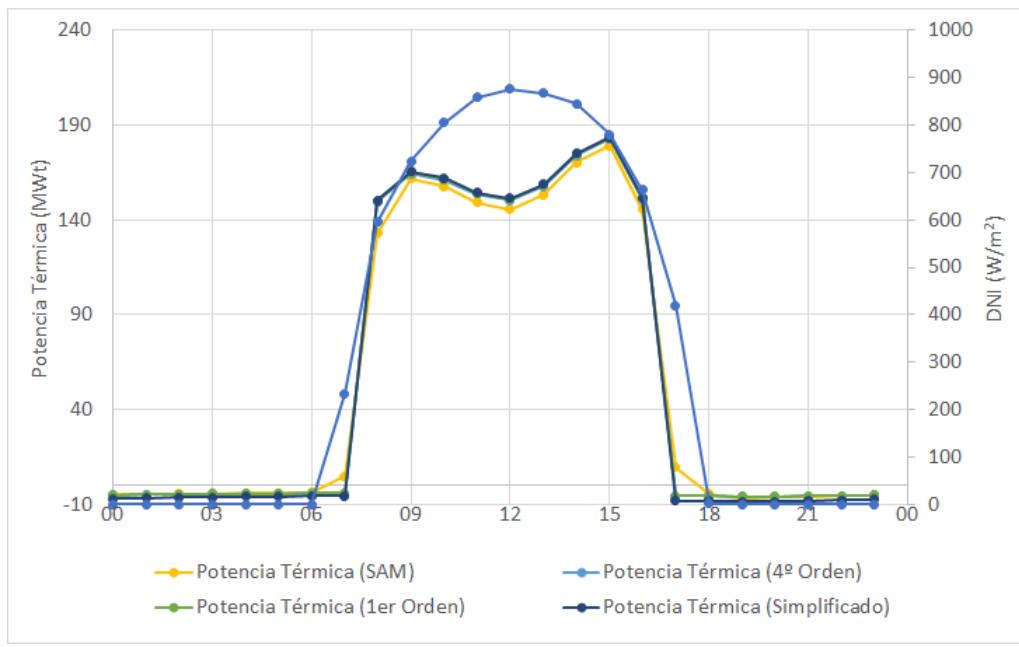


Fig. 4.6. Potencia térmica obtenida con cada uno de los tres modelos. Datos del día 2/3/2007. Condiciones estables y buena radiación

Hora	DNI (W/m <sup>2</sup> )	Caudal (SAM) (Kg/s)	Caudal (4 <sup>o</sup> Ord.) (Kg/s)	Caudal (1 <sup>er</sup> Ord.) (Kg/s)	Caudal (Simplif.) (Kg/s)
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
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 4.6. Caudales obtenidos en la simulación para cada modelo. Datos del día 2/3/2007. Condiciones estables y buena radiación

Hora	DNI (W/m <sup>2</sup> )	$P_{th}$ (SAM) (MWt)	$P_{th}$ (4 <sup>o</sup> Ord.) (MWt)	$P_{th}$ (1 <sup>er</sup> 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
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

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

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

#### 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<sup>er</sup> 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.

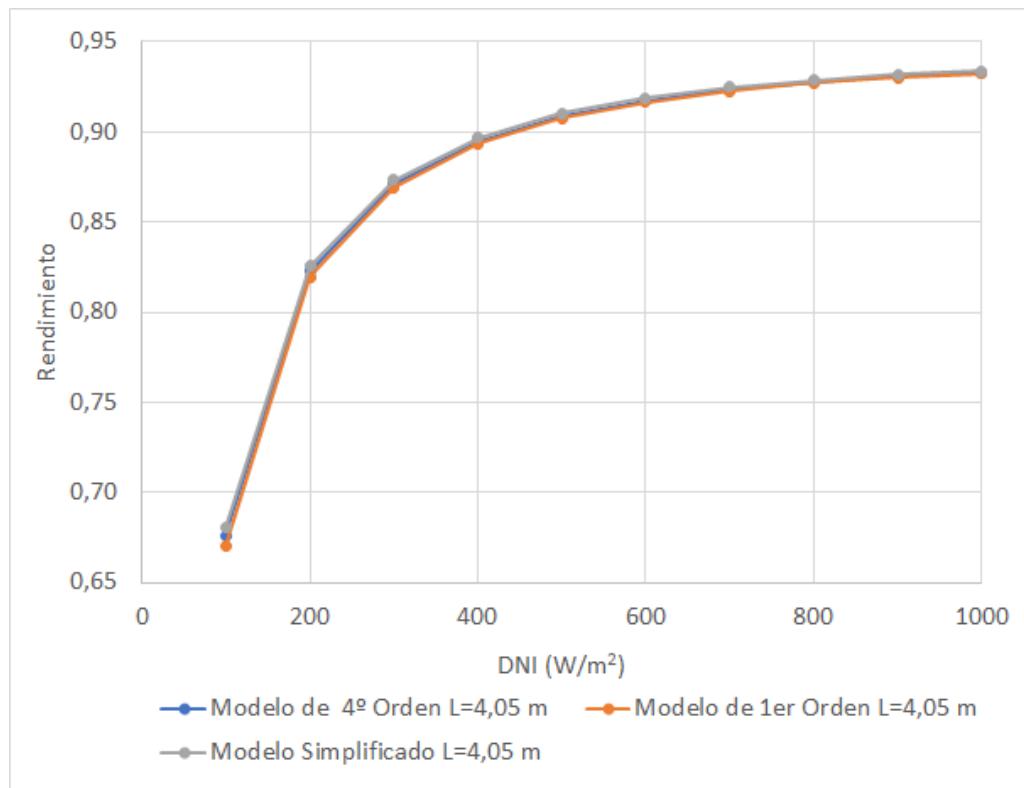


Fig. 4.7. Rendimiento calculado con cada modelo para un tamaño de malla de integración de 4,05 m

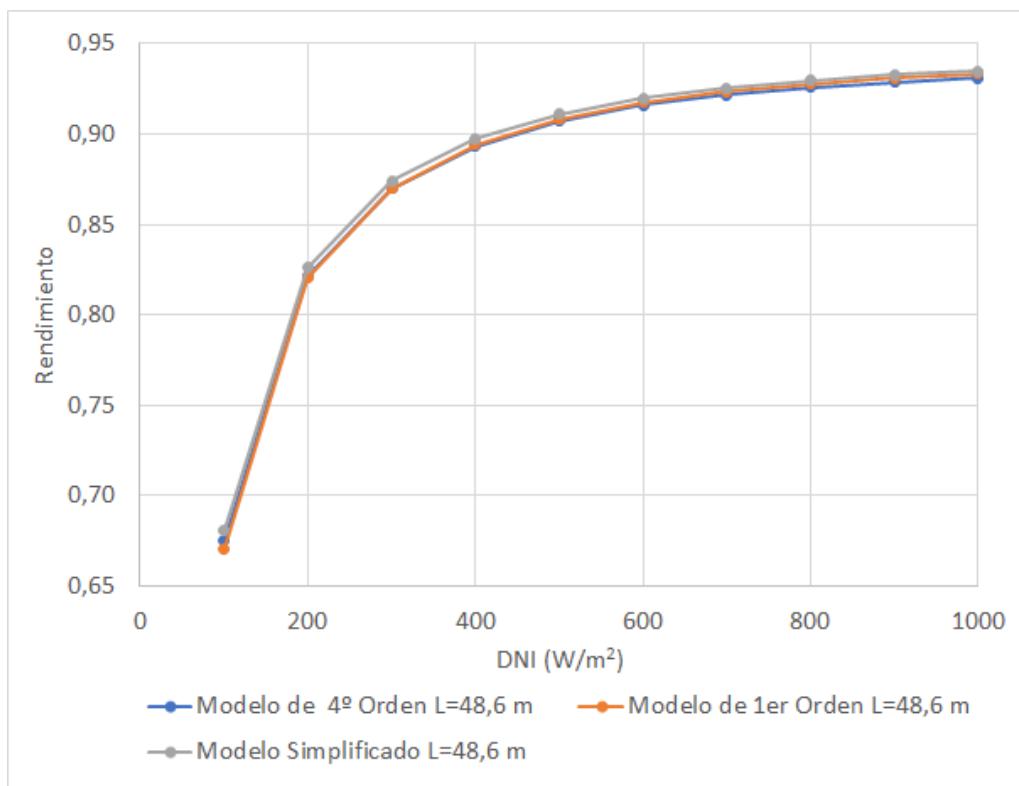


Fig. 4.8. Rendimiento calculado con cada modelo para un tamaño de malla de integración de 48,60 m

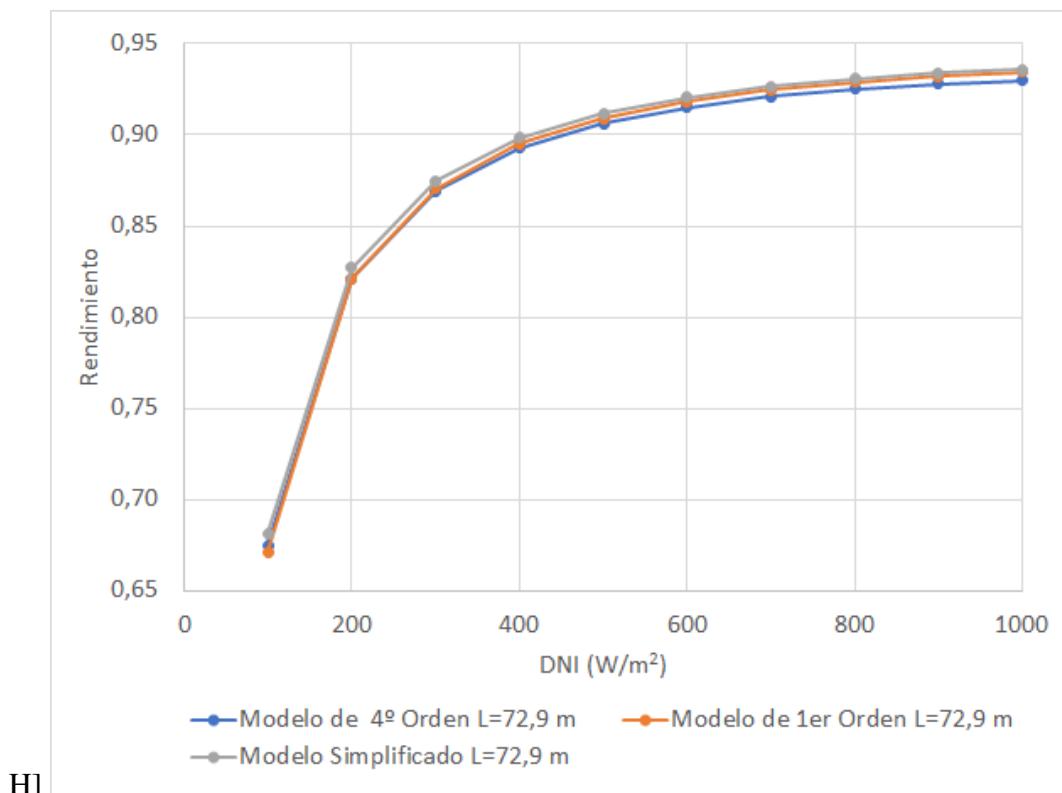


Fig. 4.9. Rendimiento calculado con cada modelo para un tamaño de malla de integración de 72,90 m

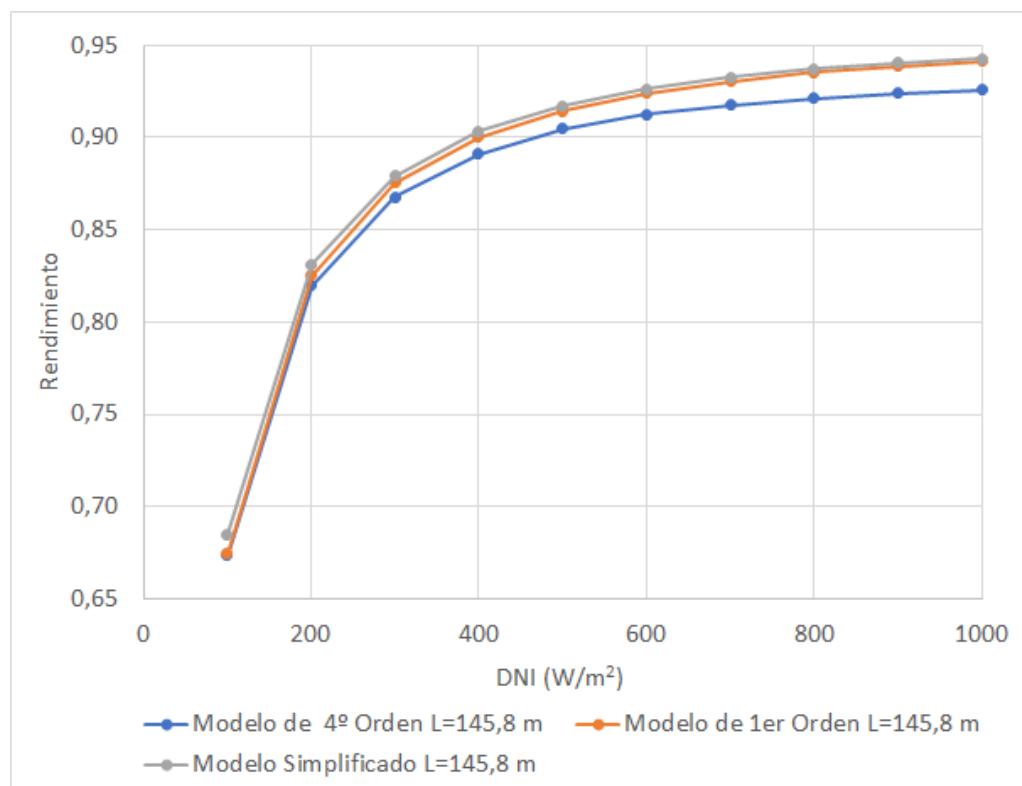


Fig. 4.10. Rendimiento calculado con cada modelo para un tamaño de malla de integración de 145,80 m

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.

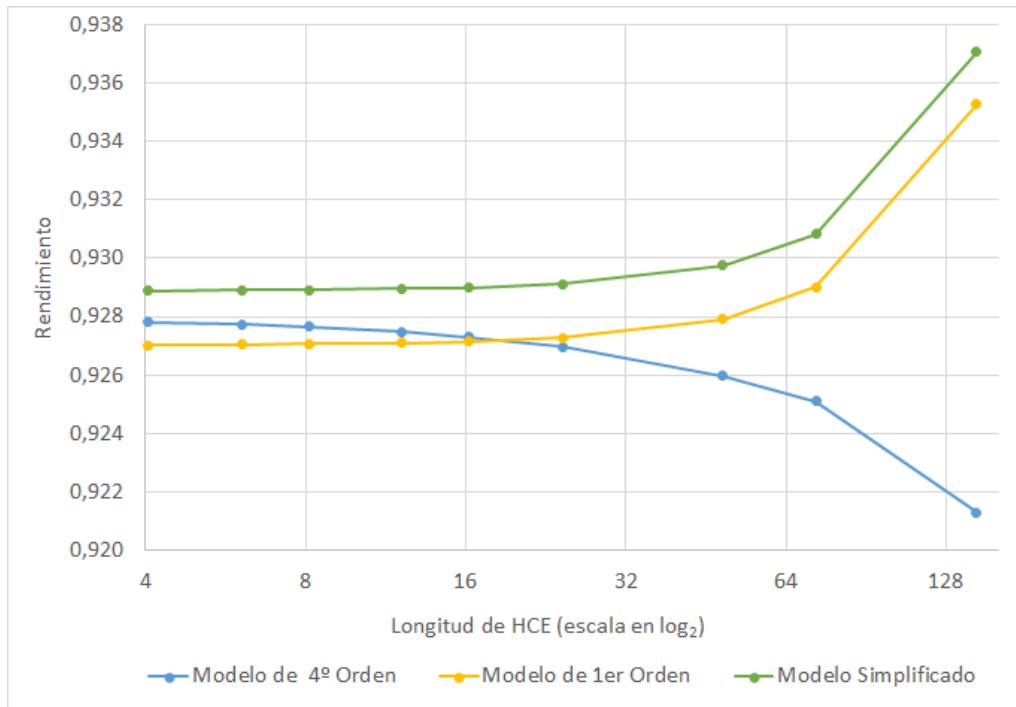


Fig. 4.11. Rendimiento calculado para diferentes tamaños de malla de integración.

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

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).

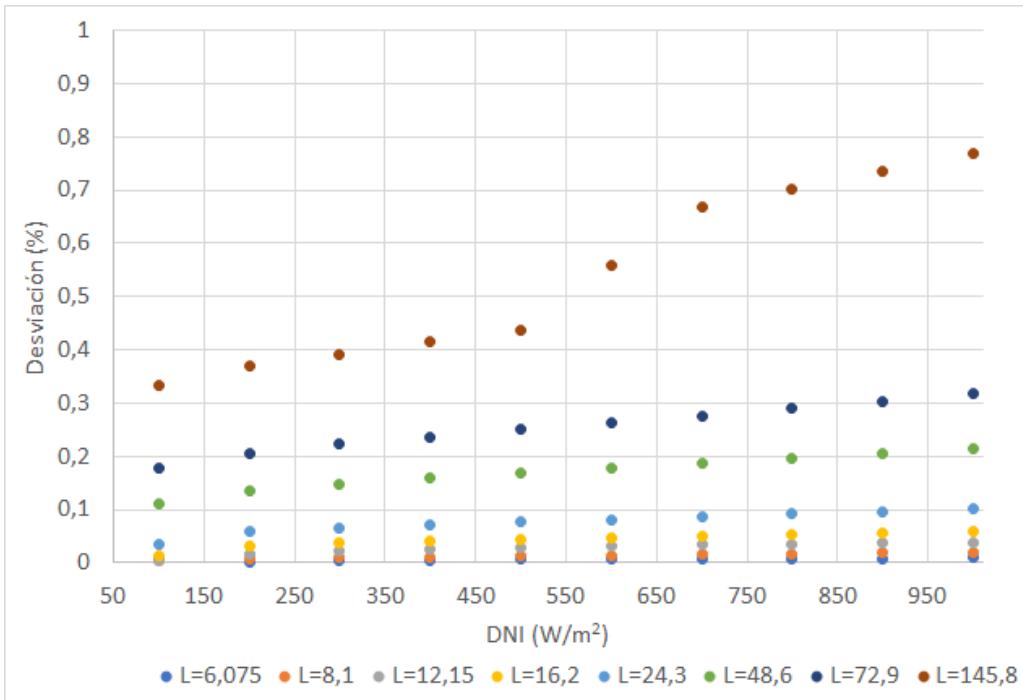


Fig. 4.12. Desviación, para simulaciones con el Modelo de 4º Orden, según diferentes tamaños de malla. Los valores muestran la desviación, en tanto por ciento, respecto a la simulación con una malla de 4,05 m.  $T_{in} = 300^{\circ}C$ ,  $\dot{m} = 6kg/s$

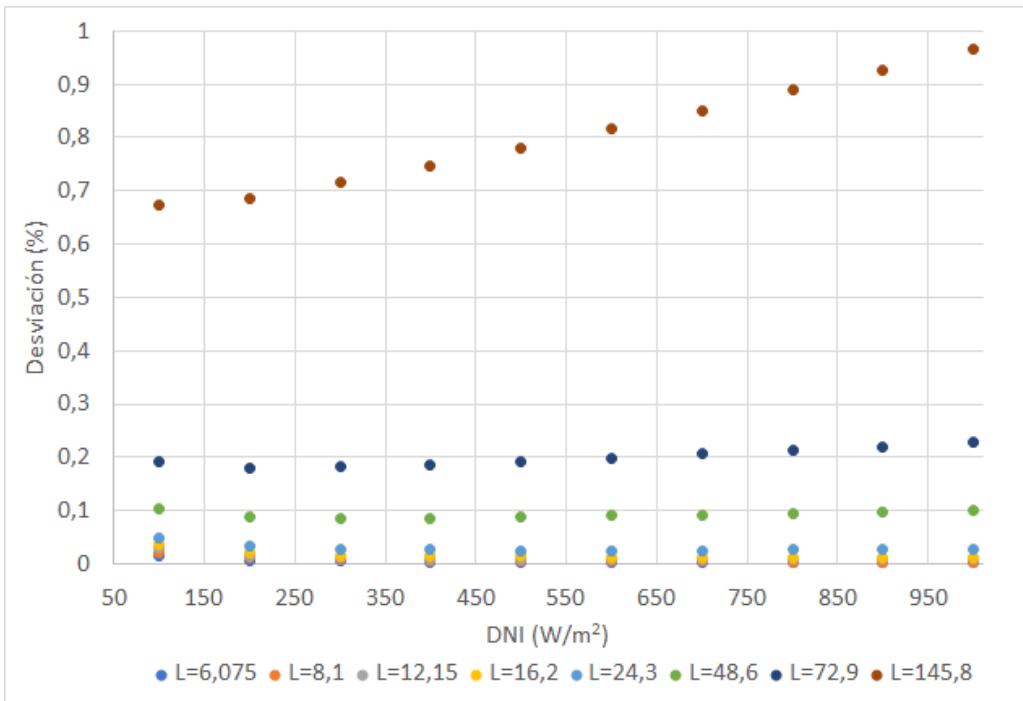


Fig. 4.13. Desviación, para simulaciones con el Modelo de 1<sup>er</sup> Orden, según diferentes tamaños de malla. Los valores muestran la desviación, en tanto por ciento, respecto a la simulación con una malla de 4,05 m.  $T_{in} = 300^{\circ}C$ ,  $\dot{m} = 6kg/s$

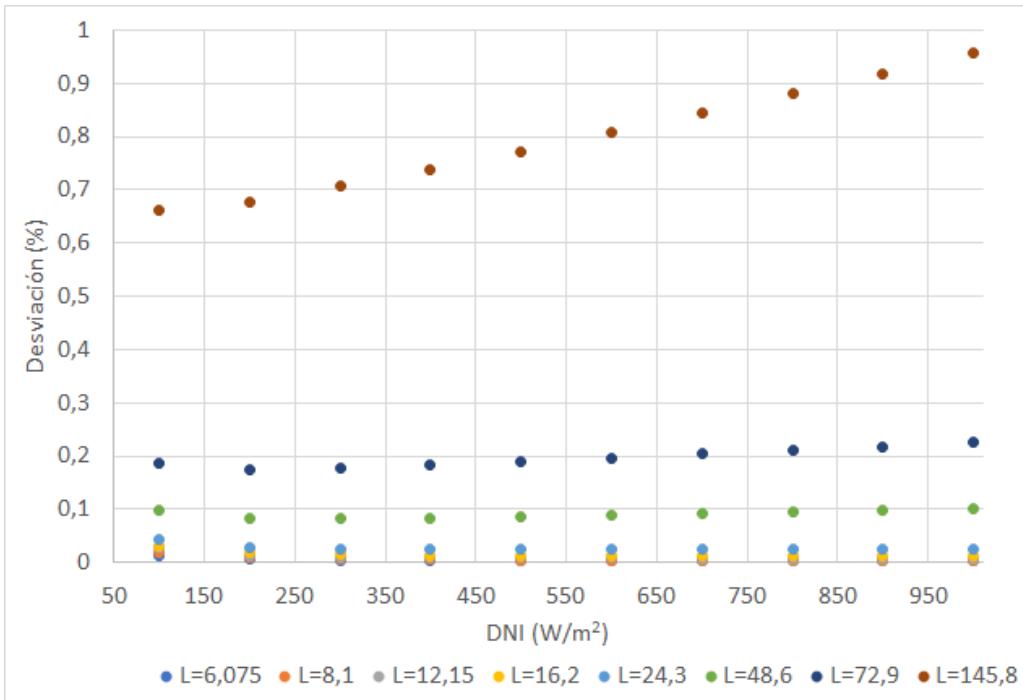


Fig. 4.14. Desviación, para simulaciones con el Modelo de Simplificado, según diferentes tamaños de malla. Los valores muestran la desviación, en tanto por ciento, respecto a la simulación con una malla de 4,05 m.  $T_{in} = 300^{\circ}C$ ,  $\dot{m} = 6kg/s$

#### 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 ( $39,1^{\circ}N$   $3,16^{\circ}W$ ) 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.

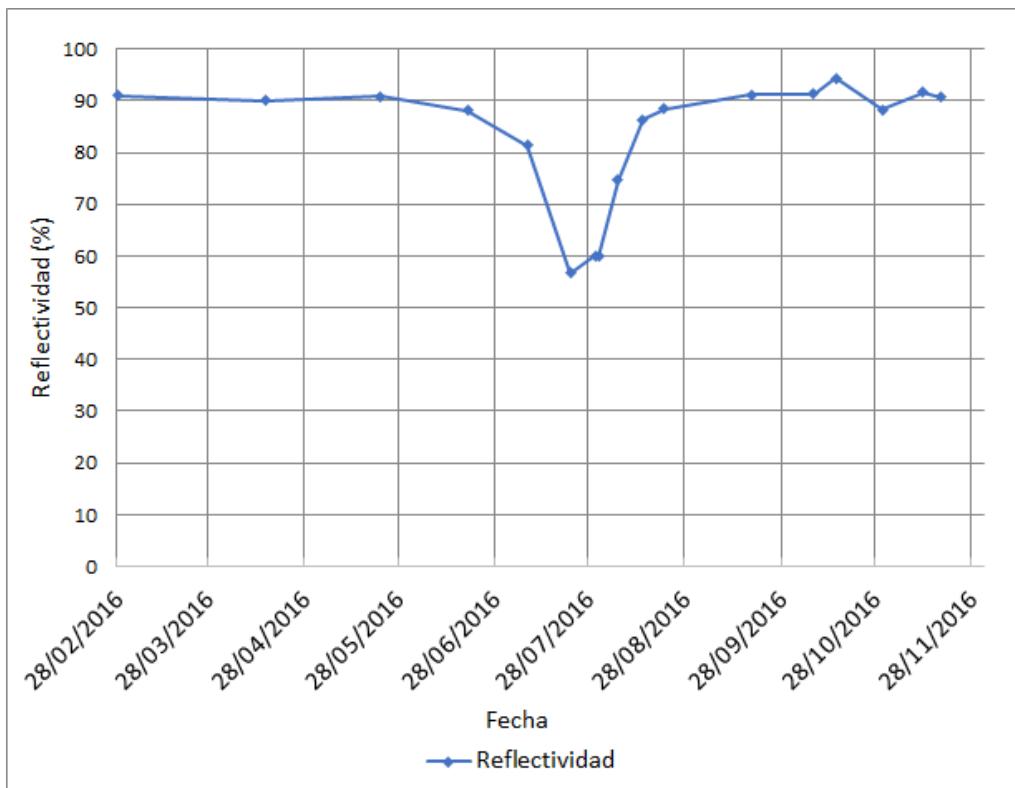


Fig. 4.16. Reflectividad registrada durante el mantenimiento. Fuente: Propia

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.

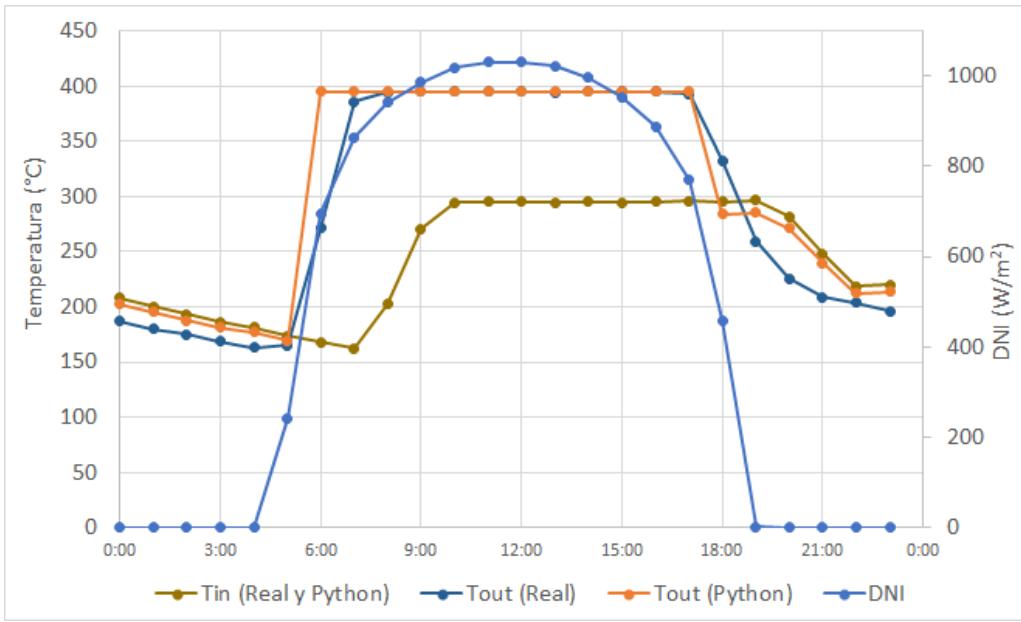


Fig. 4.17. Temperaturas de operación reales y simuladas para el día 1/5/2016. Condiciones estables y buena radiación

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.

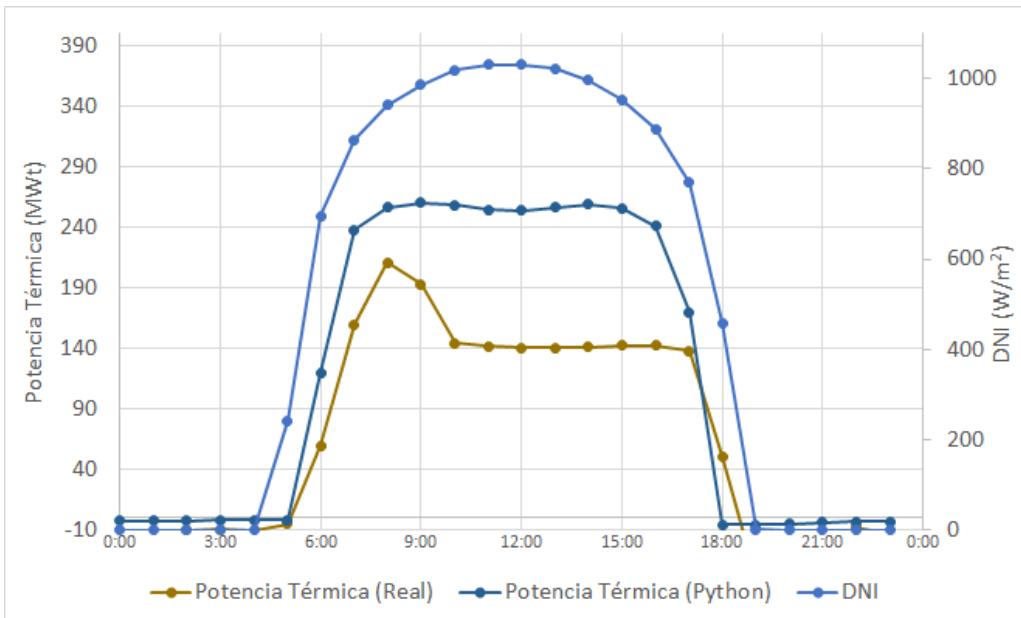


Fig. 4.18. Potencia térmica real y simulada para el día 1/5/2016. Condiciones estables y buena radiación

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.

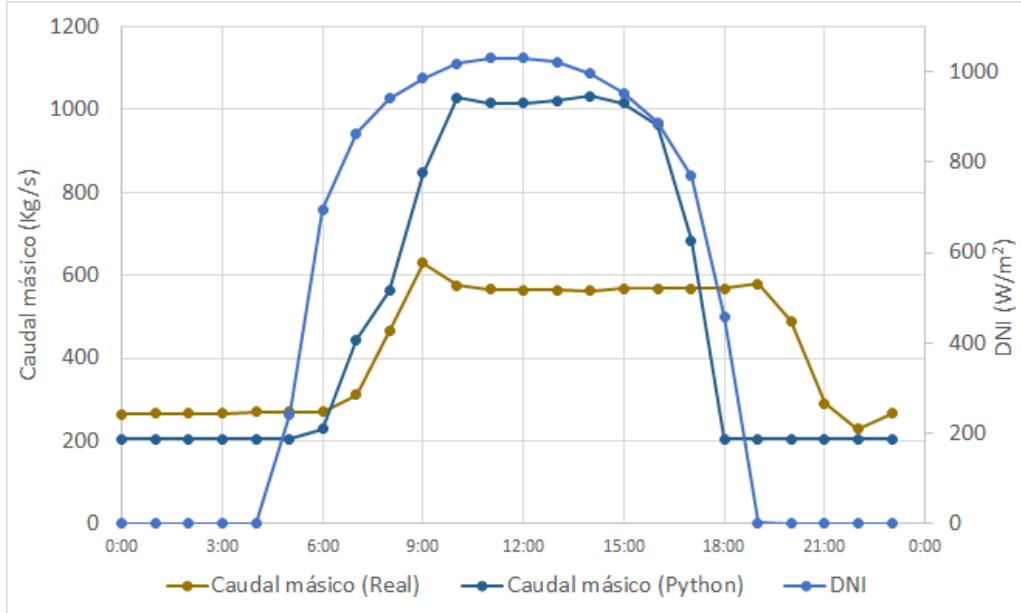


Fig. 4.19. Caudal real y simulado para el día 1/5/2016. Condiciones estables y buena radiación

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  $700 \text{ W/m}^2$ , la temperatura de entrada y salida del campo están muy cerca de las nominales ( $T_{in} > 290\text{C}$  y  $T_{out} > 390\text{C}$ ).

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.???. 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.

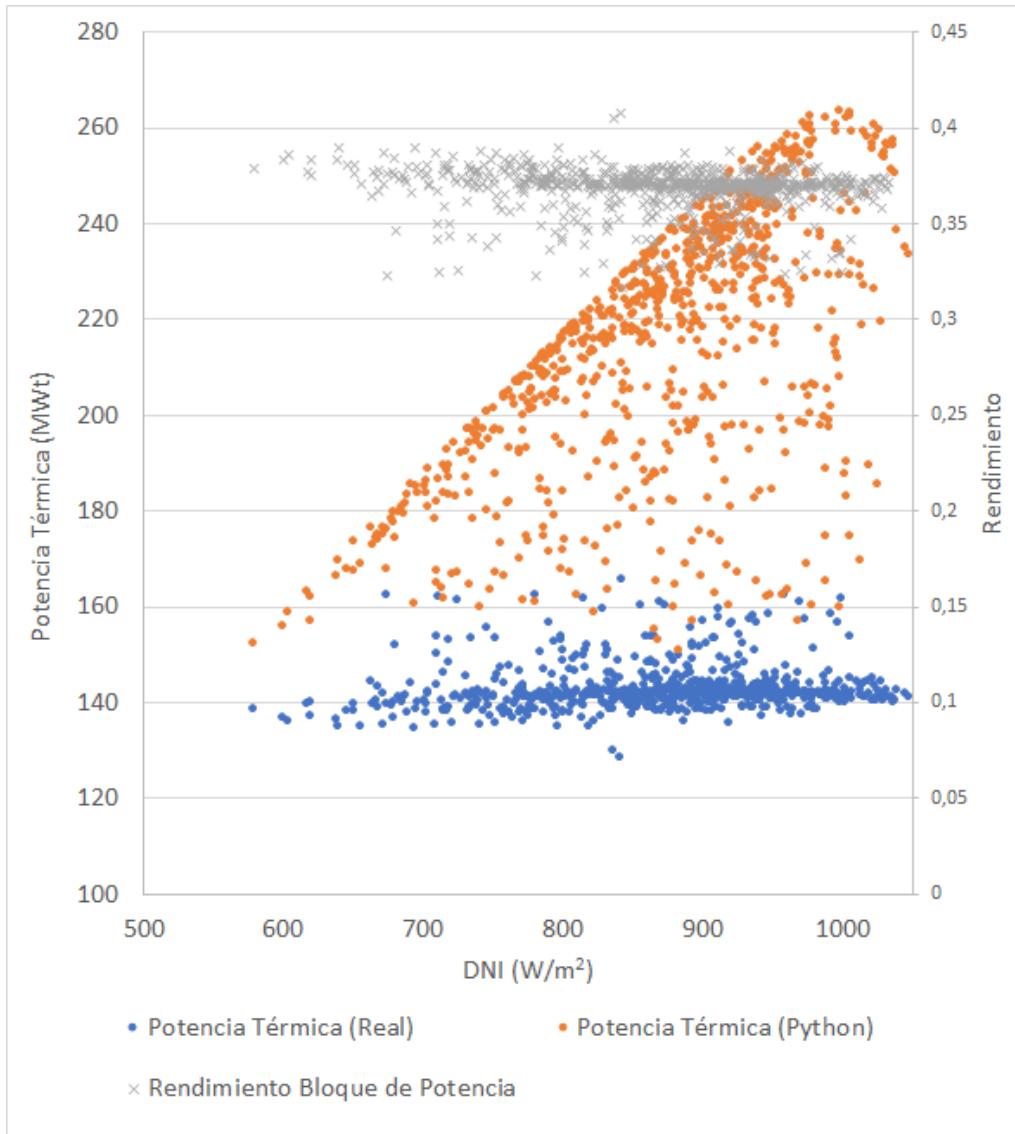


Fig. 4.20. Potencia térmica real y simulada en función de DNI cuando la planta trabaja en condiciones nominales ( $T_{in}>290\text{ }^{\circ}\text{C}$ ,  $T_{out}>390\text{ }^{\circ}\text{C}$ ). En el eje vertical derecho se representa el rendimiento global del bloque de potencia

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**

Conclusiones y trabajo futuro.

## 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] 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).
- [3] *Protermosolar*, es, <https://www.protermosolar.com/>.
- [4] A. Patnode, “Simulation and Performance Evaluation of Parabolic Trough Solar Power Plants,” Mechanical Engineering, University of Wisconsin-Madison, 2006.
- [5] V. E. Dudley et al., “Test Results: SEGS LS-2 Solar Collector,” Sandia National Laboratories, Test Result SAND94-1884, dic. de 1994.
- [6] 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.
- [7] 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).
- [8] 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.
- [9] 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).

- [10] 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).
- [11] J. A. Duffie y W. A. Beckman, *Solar Engineering of Thermal Processes*, en, Third. John Wiley & Sons, 2006.
- [12] 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.<sup>o</sup> 1, p. 011012, feb. de 2012. doi: [10.1115/1.4005247](https://doi.org/10.1115/1.4005247).
- [13] 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).
- [14] E. Zarza, *Apuntes Del Master Consultor En Energías Renovables*, 2006.
- [15] 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).
- [16] 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.
- [17] 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.<sup>o</sup> 29, p. 884, sep. de 2018. doi: [10.21105/joss.00884](https://doi.org/10.21105/joss.00884).
- [18] 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).
- [19] M. Machado, “A Product Technical Data DOWTHERM A Heat Transfer Fluid,” en,
- [20] 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).

## ANEXO A. GLOSARIO

### Acrónimos

CCP	Colector Cilindro-Parabólico
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)
IPH	Industrial Process Heat
SAM	System Advisor Model
SCA	Solar Collector Assembly
SCE	Solar Collector Element
TFG	Trabajo de Fin de Grado
UNED	Universidad Nacional de Educación a Distancia

### Símbolos latinos

$C_g$	factor de concentración
$C_p$	Calor específico a presión constante, ( $J/Kg \cdot K$ )
$D_{ri}$	diámetro interior del tubo absorbedor (m)
$D_{ro}$	diámetro exterior del tubo absorbedor (m)
$F'_{crit}$	
$H$	Entalpía específica, ( $J/Kg$ )
$h_{ext}$	<i>coeficiente de transferencia de calor convectivo equivalente</i>
$h_{int}$	coeficiente de transferencia de calor convectivo hacia el interior
$k_T$	Conductividad térmica, ( $W/m \cdot K$ )
$L$	longitud del tubo absorbedor (m)
$\dot{q}_{abs}''$	
$\dot{q}_{perd}''(x)$	
$\dot{q}_u''(x)$	

$T_f$  temperatura del fluido

$T_{ro}(x)$ temperatura de pared exterior

$U_{crit}$

$U_{rec}$  coeficiente global de transferencia de calor hacia el interior ( $W/m^2 \cdot K$ )

### Símbolos griegos

$\varepsilon_{ext}$  emisividad equivalente de la superficie exterior del tubo

$\eta_{opt}(\theta)$ rendimiento óptico

$\eta_T$  rendimiento para la totalidad del receptor en el modelo simplificado

$\eta(x)$  rendimiento integral hasta una distancia x de la entrada

$\eta_{sombra}$ coeficiente de pérdidas por sombreado

$\eta_{bordes}$ coeficiente de pérdidas geométricas

$\gamma_L$  factor de longitud efectiva

$\gamma_g$  factor de interceptación geométrico

$\rho(T)$  densidad ( $kg/m^3$ )

$\sigma$  constante de Stefan-Boltzmann ( $5.67 \times 10^{-8} W/m^2 K^4$ )

$\mu(T)$  Viscosidad dinámica, ( $Pa \cdot m$ )

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

```

```

361
362     # Ec. 3.63
363     fcrit = 1 / (1 + (ucrit / urec))
364
365     # Ec. 3.64
366     Aext = np.pi * dro * x # Pendiente de confirmar
367     NTUperd = ucrit * Aext / (massflow * cp)
368
369     if qabs > qcrit:
370         hce.pr = ((1 - (qcrit / qabs)) *
371                     (1 / (NTUperd * x)) *
372                     (1 - np.exp(-NTUperd * fcrit * x)))
373         hce.pr = hce.pr * (1 - qlost_brackets / qabs)
374         hce.qlost = qlost
375         hce.qlost_brackets = qlost_brackets
376         hce.set_tout(htf)
377         hce.set_pout(htf)
378
379     else:
380         hce.pr = 0.0
381         errtro = 10.0
382         tf = 0.5 * (hce.tin + hce.tout)
383         tro1 = tf - 5
384         while (errtro > self.max_err_tro):
385
386             kt = htf.get_thermal_conductivity(tro1, hce.pin)
387
388             fx = lambda tro1: ((2 * np.pi * kt * (tf - tro1) /
389                             np.log(dro/dri)) -
390                             sigma * hce.get_eext(tro1, wspd) *
391                             (tro1**4 - text**4) - hce.get_hext(wspd) -
392                             hce.get_qlost_brackets(tro1, text))
393
394             root = sc.optimize.newton(fx,
395                                     tro1,
396                                     maxiter=100000)
397
398             tro2 = root
399             eext = hce.get_eext(tro2, wspd)
400             # External Convective Heat Transfer equivalent coefficient
401             hext = hce.get_hext(wspd)
402
403             qlost = sigma * eext * (tro2**4 - text**4) + \
404                   hext * (tro2 - text)
405
406             # Thermal power lost through brackets
407             qlost_brackets = hce.get_qlost_brackets(tro2, text)
408
409             hce.qlost = qlost
410             hce.qlost_brackets = qlost_brackets
411             hce.set_tout(htf)
412             hce.set_pout(htf)
413             tf = 0.5 * (hce.tin + hce.tout)
414             errtro = abs(tro2 - tro1)
415             tro1 = tro2
416
417
418 class ModelBarberoSimplified(Model):
419
420     def __init__(self, settings):

```

```

421
422     self.parameters = settings
423     self.max_err_t = self.parameters['max_err_t']
424
425     def calc_pr(self, hce, htf, row, qabs=None):
426
427         if qabs is None:
428             qabs = hce.qabs
429
430         # If the hce is the first one in the loop tf = tin, else
431         # tf equals tin plus half the jump of temperature in the previous hce
432         if hce.hce_order == 0:
433             tf = hce.tin # HTF bulk temperature
434         else:
435             tf = hce.tin + 0.5 * (hce.sca.hces[hce.hce_order - 1].tout -
436                                   hce.sca.hces[hce.hce_order - 1].tin)
437
438         wspd = row[1]['Wspd'] # Wind speed
439         text = row[1]['DryBulb'] # Dry bulb ambient temperature
440         sigma = sc.constants.sigma # Stefan-Bolztmann constant
441         x = 1 # Calculation grid fits hce longitude
442
443         # Specific Capacity
444         cp = htf.get_specific_heat(tf, hce.pin)
445
446         # Internal heat transfer coefficient. Eq. 3.22 Barbero2016
447         urec = hce.get_urec(tf, hce.pin, htf)
448
449         # We suppose performance, pr = 1, at first
450         pr = 1.0
451         tro = tf + qabs * pr / urec
452
453         # HCE emittance
454         eext = hce.get_eext(tro, wspd)
455         # External Convective Heat Transfer equivalent coefficient
456         hext = hce.get_hext(wspd)
457
458         # Thermal power lost through brackets
459         qlost_brackets = hce.get_qlost_brackets(tf, text)
460
461         # Thermal power loss. Eq. 3.23 Barbero2016
462         qlost = sigma * eext * (tro**4 - text**4) + hext * (tro - text) + \
463                 qlost_brackets
464
465         # Critical Thermal power loss. Eq. 3.50 Barbero2016
466         qcrit = sigma * eext * (tf**4 - text**4) + hext * (tf - text)
467
468         # Critical Internal heat transfer coefficient, Eq. 3.51 Barbero2016
469         ucrit = 4 * sigma * eext * tf**3 + hext
470
471         # Ec. 3.63
472         ## fcrit = (1 / ((4 * eext * tfe**3 / urec) + (hext / urec) + 1))
473         fcrit = 1 / (1 + (ucrit / urec))
474
475         if qabs > qcrit:
476
477             hce.pr = fcrit * (1 - (qcrit / qabs))
478
479         else:
480             hce.pr = 0

```

```

481
482     hce.qlost = qlost
483     hce.qlost_brackets = qlost_brackets
484     hce.set_tout(htf)
485     hce.set_pout(htf)
486
487
488 class HCE(object):
489
490     def __init__(self, sca, hce_order, settings):
491
492         self.sca = sca # SCA in which the HCE is located
493         self.hce_order = hce_order # Relative position of the HCE in the SCA
494         self.parameters = settings # Set of parameters of the HCE
495         self.tin = 0.0 # Temperature of the HTF when enters in the HCE
496         self.tout = 0.0 # Temperature of the HTF when goes out the HCE
497         self.pin = 0.0 # Pressure of the HTF when enters in the HCE
498         self.pout = 0.0 # Pressure of the HTF when goes out the HCE
499         self.pr = 0.0 # Thermal performance of the HCE
500         self.pr_opt = 0.0 # Optical performance of the HCE+SCA set
501         self.qabs = 0.0 # Thermal which reach the aboserber tube
502         self.qlost = 0.0 # Thermal power lost through out the HCE
503         self.qlost_brackets = 0.0 # Thermal power lost in brackets and arms
504
505     def set_tin(self):
506
507         if self.hce_order > 0:
508             self.tin = self.sca.hces[self.hce_order-1].tout
509         elif self.sca.sca_order > 0:
510             self.tin = self.sca.loop.scas[self.sca.sca_order-1].hces[-1].tout
511         else:
512             self.tin = self.sca.loop.tin
513
514     def set_pin(self):
515
516         if self.hce_order > 0:
517             self.pin = self.sca.hces[self.hce_order-1].pout
518         elif self.sca.sca_order > 0:
519             self.pin = self.sca.loop.scas[self.sca.sca_order-1].hces[-1].pout
520         else:
521             self.pin = self.sca.loop.pin
522
523     def set_tout(self, htf):
524
525         if self.pr > 0:
526
527             q = (self.qabs * np.pi * self.pr *
528                  self.parameters['Length'] *
529                  self.parameters['Bellows ratio'] *
530                  self.parameters['Shield shading'] *
531                  self.parameters['Absorber tube outer diameter'])
532
533         else:
534             q = (-1 * (self.qlost) * np.pi *
535                  self.parameters['Length'] *
536                  self.parameters['Bellows ratio'] *
537                  self.parameters['Shield shading'] *
538                  self.parameters['Absorber tube outer diameter'])
539
540         self.tout = htf.get_temperature_by_integration()

```

```

541         self.tin, q, self.sca.loop.massflow, self.pin)
542
543
544     def set_pout(self, htf):
545         ...
546         TO-DO: CÁLCULO PERDIDA DE CARGA:
547
548         Ec. Colebrook-White para el cálculo del factor de fricción de Darcy
549         re_turbulent = 2300
550
551         k = self.parameters['Inner surface roughness']
552         D = self.parameters['Absorber tube inner diameter']
553         re = htf.get_Reynolds(D, self.tin, self.pin, self.sca.loop.massflow)
554
555         if re < re_turbulent:
556             darcy_factor = 64 / re
557
558         else:
559             # a = (k / D) / 3.7
560             a = k / 3.7
561             b = 2.51 / re
562             x = 1
563
564             fx = lambda x: x + 2 * np.log10(a + b * x)
565
566             dfx = lambda x: 1 + (2 * b) / (np.log(10) * (a + b * x))
567
568             root = sc.optimize.newton(fx,
569                                     x,
570                                     fprime=dfx,
571                                     maxiter=10000)
572
573             darcy_factor = 1 / (root**2)
574
575
576             rho = htf.get_density(self.tin, self.pin)
577             v = 4 * self.sca.loop.massflow / (rho * np.pi * D**2)
578             g = sc.constants.g
579
580             # Ec. Darcy-Weisbach para el cálculo de la pérdida de carga
581             deltap_mcl = darcy_factor * (self.parameters['Length'] / D) * \
582                 (v**2 / (2 * g))
583             deltap = deltap_mcl * rho * g
584             self.pout = self.pin - deltap
585             ...
586
587             self.pout = self.pin
588
589     def set_pr_opt(self, solarpos):
590
591         IAM = self.sca.get_IAM(solarpos)
592         aoi = self.sca.get_aoi(solarpos)
593         pr_opt_peak = self.get_pr_opt_peak()
594         self.pr_opt = pr_opt_peak * IAM * np.cos(np.radians(aoi))
595
596     def set_qabs(self, aoi, solarpos, row):
597         """Total solar power that reach de absorber tube per longitude unit."""
598         dni = row[1]['DNI']
599         cg = (self.sca.parameters['Aperture'] /
600               (np.pi * self.parameters['Absorber tube outer diameter']))

```

```

601
602     pr_shadows = self.get_pr_shadows2(solarpos)
603     pr_borders = self.get_pr_borders(aoi)
604     # Ec. 3.20 Barbero
605     self.qabs = self.pr_opt * cg * dni * pr_borders * pr_shadows
606
607 def get_krec(self, t):
608
609     # From Choom S. Kim for A316
610     # kt = 100*(0.1241+0.00003279*t)
611
612     # Ec. 4.22 Conductividad para el acero 321H.
613     kt = 0.0153 * (t - 273.15) + 14.77
614
615     return kt
616
617 def get_urec(self, t, p, htf):
618
619     # HCE wall thermal conductivity
620     krec = self.get_krec(t)
621
622     # Specific Capacity
623     cp = htf.get_specific_heat(t, p)
624
625     # Internal transmission coefficient.
626     hint = self.get_hint(t, p, htf)
627
628     # Internal heat transfer coefficient. Eq. 3.22 Barbero2016
629     return (1 / ((1 / hint) + (
630         self.parameters['Absorber tube outer diameter'] *
631         np.log(self.parameters['Absorber tube outer diameter']) /
632         self.parameters['Absorber tube inner diameter'])) /
633         (2 * krec)))
634
635 def get_pr_opt_peak(self):
636
637     alpha = self.get_absorptance()
638     tau = self.get_transmittance()
639     rho = self.sca.parameters['Reflectance']
640     gamma = self.sca.get_solar_fraction()
641
642     pr_opt_peak = alpha * tau * rho * gamma
643
644     if pr_opt_peak > 1 or pr_opt_peak < 0:
645         print("ERROR", pr_opt_peak)
646
647     return pr_opt_peak
648
649 def get_pr_borders(self, aoi):
650
651     if aoi > 90:
652         pr_borders = 0.0
653
654     else:
655         sca_unused_length = (self.sca.parameters["Focal Len"] *
656                               np.tan(np.radians(aoi)))
657
658         unused_hces = sca_unused_length // self.parameters["Length"]
659
660         unused_part_of_hce = ((sca_unused_length %

```

```

661                         self.parameters["Length"]) /
662                         self.parameters["Length"])
663
664         if self.hce_order < unused_hces:
665             pr_borders = 0.0
666
667     elif self.hce_order == unused_hces:
668         pr_borders = 1 - \
669             ((sca_unused_length % self.parameters["Length"]) /
670             self.parameters["Length"]))
671     else:
672         pr_borders = 1.0
673
674     if pr_borders > 1.0 or pr_borders < 0.0:
675         print("ERROR pr_bordes out of limits", pr_borders)
676
677     return pr_borders
678
679 def get_pr_shadows(self, solarpos):
680
681     if solarpos['elevation'][0] < 0:
682         shading = 1
683
684     else:
685         shading = 1 - (np.sin(np.radians(abs(solarpos['elevation'][0]))) * \
686                         self.sca.loop.parameters['row_spacing'] / \
687                         self.sca.parameters['Aperture']))
688
689     if shading < 0.0 or shading > 1.0:
690         shading = 0.0
691
692     shading = 1 - shading
693
694     if shading > 1 or shading < 0:
695         print("ERROR shading", shading)
696
697     s2 = self.get_pr_shadows2(solarpos)
698
699     return shading
700
701 def get_pr_shadows2(self, solarpos):
702
703     beta0 = self.sca.get_tracking_angle(solarpos)
704
705     if beta0 >= 0:
706         sigmabeta = 0
707     else:
708         sigmabeta = 1
709
710     # Surface tilt
711     beta = beta0 + 180 * sigmabeta
712
713     surface_azimuth = self.sca.get_surface_azimuth(solarpos)
714
715     Ls = abs(len(self.sca.loop.scas) * self.sca.parameters['SCA Length'] - \
716              abs(self.sca.loop.parameters['row_spacing']) * \
717              np.tan(np.radians(surface_azimuth - \
718                      solarpos['azimuth'][0]))))
719
720     ls = Ls / (len(self.sca.loop.scas) * self.sca.parameters['SCA Length'])

```

```

721
722     if solarpes['zenith'][0] < 90:
723         shading = min(abs(np.cos(np.radians(beta0))) * \
724                         self.sca.loop.parameters['row_spacing'] / \
725                         self.sca.parameters['Aperture'], 1)
726     else:
727         shading = 0
728
729     return shading
730
731 def get_hext(self, wspd):
732
733     # TO-DO:
734     return 0.0
735
736 def get_hint(self, t, p, fluid):
737
738     # Gnielinski correlation. Eq. 4.15 Barbero2016
739     kf = fluid.get_thermal_conductivity(t, p)
740     dri = self.parameters['Absorber tube inner diameter']
741
742     # Prandtl number
743     prf = fluid.get_prandtl(t, p)
744
745     # Reynolds number for absorber tube inner diameter, dri
746     redri = fluid.get_Reynolds(dri, t, p, self.sca.loop.massflow)
747
748     # We suppose inner wall temperature is equal to fluid temperature
749     prri = prf
750
751     # Skin friction coefficient
752     cf = np.power(1.58 * np.log(redri) - 3.28, -2)
753     nug = ((0.5 * cf * prf * (redri - 1000)) /
754             (1 + 12.7 * np.sqrt(0.5 * cf) * (np.power(prf, 2/3) - 1))) * \
755             np.power(prf / prri, 0.11)
756
757     # Internal transmission coefficient.
758     hint = kf * nug / dri
759
760     return hint
761
762 def get_eext(self, tro, wspd):
763
764     # Eq. 5.2 Barbero. Parameters given in Pg. 245
765     eext = (self.parameters['Absorber emittance factor A0'] +
766             self.parameters['Absorber emittance factor A1'] *
767             (tro - 273.15))
768     """
769     If wind speed is lower than 4 m/s, eext is increased up to a 1% at
770     4 m/s. As of 4 m/s forward, eext is increased up to a 2% at 7 m/s
771     """
772     if wspd < 4:
773         eext = eext * (1 + 0.01 * wspd / 4)
774
775     else:
776         eext = eext * (1 + 0.01 * (0.3333 * wspd - 0.3333))
777
778     return eext
779
780 def get_absorptance(self):

```

```

781     return self.parameters['Absorber absorptance']
782
783 def get_transmittance(self):
784     return self.parameters['Envelope transmittance']
785
786 def get_reflectance(self):
787     return self.sca.parameters['Reflectance']
788
789 def get_glost_brackets(self, tf, text):
790
791     # Ec. 4.12
792
793     n = self.parameters['Length'] / self.parameters['Brackets'] + \
794         + (self.hce_order == 0)
795     pb = 0.2032
796     acsb = 1.613e-4
797     kb = 48
798     hb = 20
799     tbase = tf - 10
800
801     L = self.parameters['Length']
802
803     return n * (np.sqrt(pb * kb * acsb * hb) * (tbase - text)) / L
804
805 def get_previous_pr(self):
806
807     if self.hce_order > 0:
808         previous_pr = self.sca.hces[self.hce_order-1].pr
809     elif self.sca.sca_order > 0:
810         previous_pr = self.sca.loop.scas[self.sca.sca_order-1].hces[-1].pr
811     else:
812         previous_pr = 1.0
813
814     return previous_pr
815
816 def get_index(self):
817
818     if hasattr(self.sca.loop, 'subfield'):
819         index = [self.sca.loop.subfield.name,
820                  self.sca.loop.loop_order,
821                  self.sca.sca_order,
822                  self.hce_order]
823     else:
824         index = ['BL',
825                  self.sca.sca_order,
826                  self.hce_order]
827
828     return index
829
830
831 class SCA(object):
832
833     def __init__(self, loop, sca_order, settings):
834
835         self.loop = loop
836         self.sca_order = sca_order
837         self.hces = []

```

```

841     self.status = 'focused'
842     self.tracking_angle = 0.0
843     self.parameters = dict(settings)
844
845
846     def get_solar_fraction(self):
847
848         if self.status == 'defocused':
849             solarfraction = 0.0
850         elif self.status == 'focused':
851
852             # Cleanliness two times because it affects mirror and envelope
853             solarfraction = (self.parameters['Geom.Accuracy'] *
854                             self.parameters['Track Twist'] *
855                             self.parameters['Cleanliness'] *
856                             self.parameters['Cleanliness'] *
857                             self.parameters['Factor'] *
858                             self.parameters['Availability']))
859         else:
860             solarfraction = 1.0
861
862         if solarfraction > 1 or solarfraction < 0:
863             print("ERROR", solarfraction)
864
865         return solarfraction
866
867     def get_IAM(self, solarpos):
868
869         if solarpos['zenith'][0] > 80:
870             kiam = 0.0
871         else:
872             aoi = self.get_aoi(solarpos)
873
874             F0 = self.parameters['IAM Coefficient F0']
875             F1 = self.parameters['IAM Coefficient F1']
876             F2 = self.parameters['IAM Coefficient F2']
877
878             if (aoi > 0 and aoi < 80):
879                 kiam = (F0 + (F1 * np.radians(aoi) + F2 * np.radians(aoi)**2) /
880                         np.cos(np.radians(aoi)))
881
882             if kiam > 1.0:
883                 kiam = 1.0
884
885             if kiam > 1.0 or kiam < 0.0:
886                 print("ERROR", kiam, aoi)
887
888         return kiam
889
890     def get_aoi(self, solarpos):
891
892         sigmabeta = 0.0
893         beta0 = 0.0
894
895         surface_azimuth = self.get_surface_azimuth(solarpos)
896         beta0 = self.get_tracking_angle(solarpos)
897
898         if beta0 >= 0:
899             sigmabeta = 0
900         else:

```

```

901     sigmabeta = 1
902
903     beta = beta0 + 180 * sigmabeta
904     aoi = pvlib.irradiance.aoi(beta,
905                                     surface_azimuth,
906                                     solarpos['zenith'][0],
907                                     solarpos['azimuth'][0])
908
909     return aoi
910
911
912 def get_tracking_angle(self, solarpos):
913
914     surface_azimuth = self.get_surface_azimuth(solarpos)
915     # Tracking angle for a collector with tilt = 0
916     # Ec. 2.32 Technical Manual for the SAM Physical Trough Model
917     beta0 = np.degrees(
918         np.arctan(np.tan(np.radians(solarpos['zenith'][0])) *
919             np.cos(np.radians(surface_azimuth -
920                     solarpos['azimuth'][0]))))
921
922     return beta0
923
924
925 def get_surface_azimuth(self, solarpos):
926
927     if self.loop.parameters['Tracking Type'] == 1: # N-S single axis
928         if solarpos['azimuth'][0] > 0 and solarpos['azimuth'][0] <= 180:
929             surface_azimuth = 90 # Surface facing east
930         else:
931             surface_azimuth = 270 # Surface facing west
932     elif self.loop.parameters['Tracking Type'] == 2: # E-W single axis
933         surface_azimuth = 180 # Surface facing the equator
934
935
936
937 class __Loop__(object):
938
939
940     def __init__(self, settings):
941
942         self.scas = []
943         self.parameters = settings
944
945         self.tin = 0.0
946         self.tout = 0.0
947         self.pin = 0.0
948         self.pout = 0.0
949         self.massflow = 0.0
950         self.qabs = 0.0
951         self.qlost = 0.0
952         self.qlost_brackets = 0.0
953         self.wasted_power = 0.0
954         self.pr = 0.0
955         self.pr_opt = 0.0
956
957         self.act_tin = 0.0
958         self.act_tout = 0.0
959         self.act_pin = 0.0
960         self.act_pout = 0.0
961         self.act_massflow = 0.0 # Actual massflow measured by the flowmeter
962
963         self.wasted_power = 0.0

```

```

961         self.tracking = True
962
963     def initialize(self, type_of_source, source=None):
964
965         if type_of_source == 'rated':
966             self.massflow = self.parameters['rated_massflow']
967             self.tin = self.parameters['rated_tin']
968             self.pin = self.parameters['rated_pin']
969
970         elif type_of_source == 'subfield' and source is not None:
971             self.massflow = source.massflow / len(source.loops)
972             self.tin = source.tin
973             self.pin = source.pin
974
975         elif type_of_source == 'solarfield' and source is not None:
976             self.massflow = source.massflow / source.total_loops
977             self.tin = solarfield.tin
978             self.pin = solarfield.pin
979
980         elif type_of_source == 'values' and source is not None:
981             self.massflow = source['massflow']
982             self.tin = source['tin']
983             self.pin = source['pin']
984
985     else:
986         print("ERROR initialize()")
987
988
989     def load_actual(self, subfield = None):
990
991         if subfield == None:
992             subfield = self.subfield
993
994         self.act_massflow = subfield.act_massflow / len(subfield.loops)
995         self.act_tin = subfield.act_tin
996         self.act_tout = subfield.act_tout
997         self.act_pin = subfield.act_pin
998         self.act_pout = subfield.act_pout
999
1000    def set_loop_values_from_HCEs(self):
1001
1002        pr_list = []
1003        qabs_list = []
1004        qlost_brackets_list = []
1005        qlost_list = []
1006        pr_opt_list = []
1007
1008        for s in self.scas:
1009            for h in s.hces:
1010                pr_list.append(h.pr)
1011                qabs_list.append(
1012                    h.qabs *
1013                    np.pi *
1014                    h.parameters['Length'] *
1015                    h.parameters['Bellows ratio'] *
1016                    h.parameters['Shield shading'] *
1017                    h.parameters['Absorber tube outer diameter'])
1018                qlost_brackets_list.append(
1019                    h.qlost_brackets *
1020                    np.pi *

```

```

1021             h.parameters['Length'] *
1022             h.parameters['Absorber tube outer diameter'])
1023         qlost_list.append(
1024             h.qlost *
1025             np.pi *
1026             h.parameters['Length'] *
1027             h.parameters['Absorber tube outer diameter'])
1028     pr_opt_list.append(h.pr_opt)
1029
1030     self.pr = np.mean(pr_list)
1031     self.qabs = np.sum(qabs_list)
1032     self.qlost_brackets = np.sum(qlost_brackets_list)
1033     self.qlost = np.sum(qlost_list)
1034     self.pr_opt = np.mean(pr_opt_list)
1035
1036 def load_from_base_loop(self, base_loop):
1037
1038     self.massflow = base_loop.massflow
1039     self.tin = base_loop.tin
1040     self.pin = base_loop.pin
1041     self.tout = base_loop.tout
1042     self.pout = base_loop.pout
1043     self.pr = base_loop.pr
1044     self.wasted_power = base_loop.wasted_power
1045     self.pr_opt = base_loop.pr_opt
1046     self.qabs = base_loop.qabs
1047     self.qlost = base_loop.qlost
1048     self.qlost_brackets = base_loop.qlost_brackets
1049
1050 def calc_loop_pr_for_massflow(self, row, solarpos, htf, model):
1051
1052     for s in self.scas:
1053         aoi = s.get_aoi(solarpos)
1054         for h in s.hces:
1055             h.set_pr_opt(solarpos)
1056             h.set_qabs(aoi, solarpos, row)
1057             h.set_tin()
1058             h.set_pin()
1059             model.calc_pr(h, htf, row)
1060
1061     self.tout = self.scas[-1].hces[-1].tout
1062     self.pout = self.scas[-1].hces[-1].pout
1063     self.set_loop_values_from_HCEs()
1064     self.set_wasted_power(htf)
1065
1066 def calc_loop_pr_for_tout(self, row, solarpos, htf, model):
1067
1068     dri = self.scas[0].hces[0].parameters['Absorber tube inner diameter']
1069     min_reynolds = self.scas[0].hces[0].parameters['Min Reynolds']
1070
1071     min_massflow = htf.get_massflow_from_Reynolds(dri, self.tin, self.pin,
1072                                                 min_reynolds)
1073
1074     max_error = model.max_err_t # % desviation tolerance
1075     search = True
1076     step = 0
1077
1078     while search:
1079
1080         self.calc_loop_pr_for_massflow(row, solarpos, htf, model)

```

```

1081     err = abs(self.tout-self.parameters['rated_tout'])
1082
1083     if err > max_error and step <1000:
1084         step += 1
1085         if self.tout >= self.parameters['rated_tout']:
1086             self.massflow *= (1 + err / self.parameters['rated_tout']))
1087             search = True
1088         elif (self.massflow > min_massflow and
1089               self.massflow >
1090               self.parameters['min_massflow']):
1091             self.massflow *= (1 - err / self.parameters['rated_tout']))
1092             search = True
1093         else:
1094             self.massflow = max(
1095                 min_massflow,
1096                 self.parameters['min_massflow'])
1097             self.calc_loop_pr_for_massflow(row, solarpos, htf, model)
1098             search = False
1099
1100     else:
1101         search = False
1102         if step>=1000:
1103             print("Massflow convergence failed")
1104
1105     self.tout = self.scas[-1].hces[-1].tout
1106     self.pout = self.scas[-1].hces[-1].pout
1107     self.set_loop_values_from_HCEs()
1108     self.wasted_power = 0.0
1109
1110
1111 def check_min_massflow(self, htf):
1112
1113     dri = self.parameters['Absorber tube inner diameter']
1114     t = self.tin
1115     p = self.pin
1116     re = self.parameters['Min Reynolds']
1117     loop_min_massflow = htf.get_massflow_from_Reynolds(
1118         dri, t, p , re)
1119
1120     if self.massflow < loop_min_massflow:
1121         print("Too low massflow", self.massflow , "<",
1122               loop_min_massflow)
1123
1124 def set_wasted_power(self, htf):
1125
1126     if self.tout > self.parameters['tmax']:
1127         self.wasted_power = self.massflow * htf.get_delta_enthalpy(
1128             self.parameters['tmax'], self.tout, self.pin, self.pout)
1129     else:
1130         self.wasted_power = 0.0
1131
1132 def get_values(self, type = None):
1133
1134     _values = {}
1135
1136     if type is None:
1137         _values = {'tin': self.tin, 'tout': self.tout,
1138                   'pin': self.pin, 'pout': self.pout,
1139                   'mf': self.massflow}
1140     elif type =='required':

```

```

1141         _values = {'tin': self.tin, 'tout': self.tout,
1142                     'pin': self.pin, 'pout': self.pout,
1143                     'req_mf': self.req_massflow}
1144     elif type == 'actual':
1145         _values = {'actual_tin': self.actual_tin, 'tout': self.tout,
1146                     'pin': self.pin, 'pout': self.pout,
1147                     'mf': self.req_massflow}
1148
1149     return _values
1150
1151 def get_id(self):
1152
1153     return 'LP.{0}.{1:03d}'.format(self.subfield.name, self.loop_order)
1154
1155 class Loop(__Loop__):
1156
1157     def __init__(self, subfield, loop_order, settings):
1158
1159         self.subfield = subfield
1160         self.loop_order = loop_order
1161
1162     super().__init__(settings)
1163
1164
1165 class BaseLoop(__Loop__):
1166
1167     def __init__(self, settings, sca_settings, hce_settings):
1168
1169         super().__init__(settings)
1170
1171         self.parameters_sca = sca_settings
1172         self.parameters_hce = hce_settings
1173
1174         for s in range(settings['scas']):
1175             self.scas.append(SCA(self, s, sca_settings))
1176             for h in range(settings['hces']):
1177                 self.scas[-1].hces.append(
1178                     HCE(self.scas[-1], h, hce_settings))
1179
1180
1181     def get_id(self, subfield = None):
1182
1183         id = ''
1184         if subfield is not None:
1185             id = 'LB.'+subfield.name
1186         else:
1187             id = 'LB.000'
1188
1189         return id
1190
1191     def get_glost_brackets(self, tf, text):
1192
1193         # Ec. 4.12
1194
1195         L = self.parameters_hce['Length']
1196         n = (L / self.parameters_hce['Brackets'])
1197         pb = 0.2032
1198         acsb = 1.613e-4
1199         kb = 48
1200         hb = 20
1201         tbase = tf - 10

```

```

1201     return n * (np.sqrt(pb * kb * acsb * hb) * (tbase - text)) / L
1202
1203
1204     def get_pr_opt_peak(self):
1205
1206         alpha = self.parameters_hce['Absorber absorptance']
1207         tau = self.parameters_hce['Envelope transmittance']
1208         rho = self.parameters_sca['Reflectance']
1209         gamma = self.get_solar_fraction()
1210
1211         pr_opt_peak = alpha * tau * rho * gamma
1212
1213         if pr_opt_peak > 1 or pr_opt_peak < 0:
1214             print("ERROR pr_opt_peak", pr_opt_peak)
1215
1216         return pr_opt_peak
1217
1218
1219     def get_pr_borders(self, aoi):
1220
1221         if aoi > 90:
1222             pr_borders = 0.0
1223
1224         else:
1225             sca_unused_length = (self.parameters_sca["Focal Len"] *
1226                                   np.tan(np.radians(aoi)))
1227
1228             pr_borders = 1 - sca_unused_length / \
1229                         (self.parameters['hces'] * self.parameters_hce["Length"])
1230
1231         if pr_borders > 1.0 or pr_borders < 0.0:
1232             print("ERROR pr_geo out of limits", pr_borders)
1233
1234         return pr_borders
1235
1236
1237     def get_pr_shadows(self, solarpos):
1238
1239         if solarpos['elevation'][0] < 0:
1240             shading = 1
1241
1242         else:
1243             shading = 1 - (np.sin(np.radians(abs(solarpos['elevation'][0]))) *
1244                             self.parameters['row_spacing'] /
1245                             self.parameters_sca['Aperture'])
1246
1247         if shading < 0.0 or shading > 1.0:
1248             shading = 0.0
1249
1250
1251         shading = 1 - shading
1252
1253
1254         if shading > 1 or shading < 0:
1255             print("ERROR shading", shading)
1256
1257         return shading
1258
1259     def get_pr_shadows2(self, solarpos):
1260

```

```

1261     beta0 = self.get_tracking_angle(solarpos)
1262
1263     if beta0 >= 0:
1264         sigmabeta = 0
1265     else:
1266         sigmabeta = 1
1267
1268     # Surface tilt
1269     beta = beta0 + 180 * sigmabeta
1270
1271     surface_azimuth = self.get_surface_azimuth(solarpos)
1272
1273     Ls = abs(len(self.scas) * self.parameters_sca['SCA Length'] -
1274             abs(self.parameters['row_spacing']) *
1275                 np.tan(np.radians(surface_azimuth -
1276                         solarpos['azimuth'][0])))
1277
1278     ls = Ls / (len(self.scas) * self.parameters_sca['SCA Length'])
1279
1280     if solarpos['zenith'][0] < 90:
1281         shading = min(abs(np.cos(np.radians(beta0))) * \
1282                       self.parameters['row_spacing'] / \
1283                         self.parameters_sca['Aperture'], 1)
1284     else:
1285         shading = 0
1286
1287     return shading
1288
1289
1290 def get_solar_fraction(self):
1291
1292     # Cleanliness two times because it affects mirror and envelope
1293     solarfraction = (self.parameters_sca['Geom.Accuracy'] *
1294                         self.parameters_sca['Track Twist'] *
1295                         self.parameters_sca['Cleanliness'] *
1296                         self.parameters_sca['Cleanliness'] *
1297                         self.parameters_sca['Factor'] *
1298                         self.parameters_sca['Availability'])
1299
1300     if solarfraction > 1 or solarfraction < 0:
1301         print("ERROR", solarfraction)
1302
1303     return solarfraction
1304
1305 def get_IAM(self, solarpos):
1306
1307     if solarpos['zenith'][0] > 80:
1308         kiam = 0.0
1309     else:
1310
1311         aoi = self.get_aoi(solarpos)
1312
1313         F0 = self.parameters_sca['IAM Coefficient F0']
1314         F1 = self.parameters_sca['IAM Coefficient F1']
1315         F2 = self.parameters_sca['IAM Coefficient F2']
1316
1317         if (aoi > 0 and aoi < 80):
1318             kiam = (F0 + (F1 * np.radians(aoi) + F2 * np.radians(aoi)**2) / \
1319                     np.cos(np.radians(aoi)))
1320

```

```

1321     if kiam > 1.0:
1322         kiam = 1.0
1323
1324     if kiam > 1.0 or kiam < 0.0:
1325         print("ERROR", kiam, aoi)
1326
1327     return kiam
1328
1329
1330 def get_aoi(self, solarpos):
1331
1332     sigmabeta = 0.0
1333     beta0 = 0.0
1334
1335     surface_azimuth = self.get_surface_azimuth(solarpos)
1336     beta0 = self.get_tracking_angle(solarpos)
1337
1338     if beta0 >= 0:
1339         sigmabeta = 0
1340     else:
1341         sigmabeta = 1
1342
1343     beta = beta0 + 180 * sigmabeta
1344     aoi = pvlib.irradiance.aoi(beta,
1345                                 surface_azimuth,
1346                                 solarpos['zenith'][0],
1347                                 solarpos['azimuth'][0])
1348
1349
1350
1351 def get_tracking_angle(self, solarpos):
1352
1353     surface_azimuth = self.get_surface_azimuth(solarpos)
1354     # Tracking angle for a collector with tilt = 0
1355     # Ec. 2.32 Technical Manual for the SAM Physical Trough Model
1356     beta0 = np.degrees(
1357         np.arctan(np.tan(np.radians(solarpos['zenith'][0])) *
1358                 np.cos(np.radians(surface_azimuth -
1359                             solarpos['azimuth'][0]))))
1360
1361
1362
1363 def get_surface_azimuth(self, solarpos):
1364
1365     if self.parameters['Tracking Type'] == 1: # N-S single axis tracker
1366         if solarpos['azimuth'][0] > 0 and solarpos['azimuth'][0] <= 180:
1367             surface_azimuth = 90 # Surface facing east
1368         else:
1369             surface_azimuth = 270 # Surface facing west
1370     elif self.parameters['Tracking Type'] == 2: # E-W single axis tracker
1371         surface_azimuth = 180 # Surface facing the equator
1372
1373
1374
1375
1376 class Subfield(object):
1377     ...
1378     Parabolic Trough Solar Field
1379     ...
1380

```

```

1381 def __init__(self, solarfield, settings):
1382     self.solarfield = solarfield
1383     self.name = settings['name']
1384     self.parameters = settings
1385     self.loops = []
1386
1387     self.tin = 0.0
1388     self.tout = 0.0
1389     self.pin = 0.0
1390     self.pout = 0.0
1391     self.massflow = 0.0
1392     self.qabs = 0.0
1393     self.qlost = 0.0
1394     self.qlost_brackets = 0.0
1395     self.wasted_power = 0.0
1396     self.pr = 0.0
1397     self.pr_opt = 0.0
1398
1399
1400     self.act_tin = 0.0
1401     self.act_tout = 0.0
1402     self.act_pin = 0.0
1403     self.act_pout = 0.0
1404     self.act_massflow = 0.0
1405     self.pr_act_massflow = 0.0
1406
1407
1408     self.rated_tin = self.solarfield.rated_tin
1409     self.rated_tout = self.solarfield.rated_tout
1410     self.rated_pin = self.solarfield.rated_pin
1411     self.rated_pout = self.solarfield.rated_pout
1412     self.rated_massflow = (self.solarfield.rated_massflow *
1413                           self.parameters['loops'] /
1414                           self.solarfield.total_loops)
1415
1416 def set_subfield_values_from_loops(self, htf):
1417
1418     self.massflow = np.sum([l.massflow for l in self.loops])
1419     self.pr = np.sum([l.pr * l.massflow for l in self.loops]) / \
1420                     self.massflow
1421     self.pr_opt = np.sum([l.pr_opt * l.massflow for l in self.loops]) / \
1422                     self.massflow
1423     self.wasted_power = np.sum([l.wasted_power for l in self.loops]) / \
1424         1000000 # From Watts to MW
1425     self.pout = np.sum([l.pout * l.massflow for l in self.loops]) / \
1426                     self.massflow
1427     self.tout = htf.get_temperature(
1428         np.sum([l.massflow *
1429                 htf.get_enthalpy(l.tout, l.pout) for l in self.loops]) / \
1430                     self.massflow, self.pout)
1431     self.qlost = np.sum([l.qlost for l in self.loops]) / 1000000
1432     self.qabs = np.sum([l.qabs for l in self.loops]) / 1000000
1433     self.qlost_brackets = np.sum([l.qlost_brackets for l in self.loops]) \
1434         / 1000000
1435
1436 def set_massflow(self):
1437
1438     self.massflow = np.sum([l.massflow for l in self.loops])
1439
1440 def set_req_massflow(self):

```

```

1441         self.req_massflow = np.sum([l.req_massflow for l in self.loops])
1442
1443     def set_wasted_power(self):
1444
1445         self.wasted_power = np.sum([l.wasted_power for l in self.loops]) / \
1446             1000000 # From Watts to MW
1447
1448     def set_pr_req_massflow(self):
1449
1450         self.pr_req_massflow = np.sum(
1451             [l.pr_req_massflow * l.req_massflow for l in self.loops]) / \
1452                 self.req_massflow
1453
1454     def set_pr_act_massflow(self):
1455
1456         self.pr_act_massflow = np.sum(
1457             [l.pr_act_massflow * l.act_massflow for l in self.loops]) / \
1458                 self.act_massflow
1459
1460     def set_pout(self):
1461
1462         self.pout = np.sum([l.pout * l.massflow for l in self.loops]) / \
1463             self.massflow
1464
1465     def set_tout(self, htf):
1466         '''
1467             Calculates HTF output temperature throughout the solar field as a
1468             weighted average based on the enthalpy of the mass flow in each
1469             loop of the solar field
1470         '''
1471
1472         self.tout = htf.get_temperature(
1473             np.sum([l.massflow * htf.get_enthalpy(l.tout, l.pout)
1474                     for l in self.loops]) / self.massflow, self.pout)
1475
1476     def initialize(self, source, values = None):
1477
1478         if source == 'rated':
1479             self.massflow = self.rated_massflow
1480             self.tin = self.rated_tin
1481             self.pin = self.rated_pin
1482             self.tout = self.rated_tout
1483             self.pout = self.rated_pout
1484
1485         elif source == 'actual':
1486             self.massflow = self.act_massflow
1487             self.tin = self.act_tin
1488             self.pin = self.act_pin
1489             self.tout = self.act_tout
1490             self.pout = self.act_pout
1491
1492         elif source == 'values' and values is not None:
1493             self.massflow = values['massflow']
1494             self.tin = values['tin']
1495             self.pin = values['pin']
1496
1497         else:
1498             print('Select source [rated|actual|values]')
1499             sys.exit()
1500
1501     def load_actual(self, row):

```

```

1501
1502     self.act_massflow = row[1][self.get_id() +'.a.mf']
1503     self.act_tin = row[1][self.get_id() +'.a.tin']
1504     self.act_pin = row[1][self.get_id() +'.a.pin']
1505     self.act_tout = row[1][self.get_id() +'.a.tout']
1506     self.act_pout = row[1][self.get_id() +'.a.pout']
1507
1508 def get_id(self):
1509
1510     return 'SB.' + self.name
1511
1512 class SolarField(object):
1513
1514     def __init__(self, subfield_settings, loop_settings, sca_settings,
1515                  hce_settings):
1516
1517         self.subfields = []
1518         self.total_loops = 0
1519
1520         self.tin = 0.0
1521         self.tout = 0.0
1522         self.pin = 0.0
1523         self.pout = 0.0
1524         self.massflow = 0.0
1525         self.qabs = 0.0
1526         self.qlost = 0.0
1527         self.qlost_brackets = 0.0
1528         self.wasted_power = 0.0
1529         self.pr = 0.0
1530         self.pr_opt = 0.0
1531         self.pwr = 0.0
1532
1533         self.act_tin = 0.0
1534         self.act_tout = 0.0
1535         self.act_pin = 0.0
1536         self.act_pout = 0.0
1537         self.act_massflow = 0.0
1538         self.act_pwr = 0.0
1539
1540         self.rated_tin = loop_settings['rated_tin']
1541         self.rated_tout = loop_settings['rated_tout']
1542         self.rated_pin = loop_settings['rated_pin']
1543         self.rated_pout = loop_settings['rated_pout']
1544         self.rated_massflow = (loop_settings['rated_massflow'] *
1545                               self.total_loops)
1546
1547     for sf in subfield_settings:
1548         self.total_loops += sf['loops']
1549         self.subfields.append(Subfield(self, sf))
1550         for l in range(sf['loops']):
1551             self.subfields[-1].loops.append(
1552                 Loop(self.subfields[-1], l, loop_settings))
1553             for s in range(loop_settings['scas']):
1554                 self.subfields[-1].loops[-1].scas.append(
1555                     SCA(self.subfields[-1].loops[-1], s, sca_settings))
1556                 for h in range (loop_settings['hces']):
1557                     self.subfields[-1].loops[-1].scas[-1].hces.append(
1558                         HCE(self.subfields[-1].loops[-1].scas[-1], h,
1559                             hce_settings))
1560

```

```

1561     # TO-DO: FUTURE WORK
1562     self.storage_available = False
1563     self.operation_mode = "subfield_heating"
1564
1565     def initialize(self, source, values = None):
1566
1567         if source == 'rated':
1568             self.massflow = self.rated_massflow
1569             self.tin = self.rated_tin
1570             self.pin = self.rated_pin
1571             self.tout = self.rated_tout
1572             self.pout = self.rated_pout
1573
1574         elif source == 'actual':
1575             self.massflow = self.act_massflow
1576             self.tin = self.act_tin
1577             self.pin = self.act_pin
1578             self.tout = self.act_tout
1579             self.pout = self.act_pout
1580
1581         elif source == 'values' and values is not None:
1582             self.massflow = values['massflow']
1583             self.tin = values['tin']
1584             self.pin = values['pin']
1585
1586         else:
1587             print('Select source [rated|actual|values]')
1588             sys.exit()
1589
1590     def load_actual(self, htf):
1591
1592         self.act_massflow = np.sum([sb.act_massflow for sb in self.subfields])
1593         self.act_pin = np.sum(
1594             [sb.act_pin * sb.act_massflow for sb in self.subfields]) / \
1595             self.act_massflow
1596         self.act_pout = np.sum(
1597             [sb.act_pout * sb.act_massflow for sb in self.subfields]) / \
1598             self.act_massflow
1599         self.act_tin = htf.get_temperature(
1600             np.sum([sb.act_massflow *
1601                 htf.get_enthalpy(sb.act_tin, sb.act_pin)
1602                 for sb in self.subfields]) / self.act_massflow,
1603                 self.act_pin)
1604         self.act_tout = htf.get_temperature(
1605             np.sum([sb.act_massflow *
1606                 htf.get_enthalpy(sb.act_tout, sb.act_pout)
1607                 for sb in self.subfields] / self.act_massflow),
1608                 self.act_pout)
1609
1610     def set_solarfield_values_from_subfields(self, htf):
1611
1612         self.massflow = np.sum([sb.massflow for sb in self.subfields])
1613         self.pr = np.sum(
1614             [sb.pr * sb.massflow for sb in self.subfields]) / self.massflow
1615         self.pr_opt = np.sum(
1616             [sb.pr_opt * sb.massflow for sb in self.subfields]) / self.massflow
1617         self.wasted_power = np.sum([sb.wasted_power for sb in self.subfields])
1618         self.pout = np.sum(
1619             [sb.pout * sb.massflow for sb in self.subfields]) / self.massflow
1620         self.tout = htf.get_temperature(

```

```

1621         np.sum([sb.massflow * htf.get_enthalpy(sb.tout, sb.pout)
1622                   for sb in self.subfields]) / self.massflow, self.pout)
1623     self.qlost = np.sum([sb.qlost for sb in self.subfields])
1624     self.qabs = np.sum([sb.qabs for sb in self.subfields])
1625     self.qlost_brackets = np.sum(
1626         [sb.qlost_brackets for sb in self.subfields])
1627
1628 def set_massflow(self):
1629
1630     self.massflow = np.sum([sb.massflow for sb in self.subfields])
1631     self.req_massflow = np.sum([sb.req_massflow for sb in self.subfields])
1632
1633 def set_req_massflow(self):
1634
1635     self.req_massflow = np.sum([sb.req_massflow for sb in self.subfields])
1636
1637 def set_wasted_power(self):
1638
1639     self.wasted_power = np.sum([sb.wasted_power for sb in self.subfields])
1640
1641 def set_pr_req_massflow(self):
1642
1643     self.pr_req_massflow = np.sum(
1644         [sb.pr_req_massflow * sb.req_massflow for sb in self.subfields]) \
1645         / self.req_massflow
1646
1647 def set_pr_act_massflow(self):
1648
1649     self.pr_act_massflow = np.sum(
1650         [sb.pr_act_massflow * sb.act_massflow for sb in self.subfields]) \
1651         / self.act_massflow
1652
1653 def set_pout(self):
1654
1655     self.pout = np.sum(
1656         [sb.pout * sb.massflow for sb in self.subfields]) \
1657         / self.massflow
1658
1659 def set_tout(self, htf):
1660     """
1661     Calculates HTF output temperature throughout the solar plant as a
1662     weighted average based on the enthalpy of the mass flow in each
1663     subfield of the solar field
1664     """
1665     self.tout = htf.get_temperature(
1666         np.sum([sb.massflow * htf.get_enthalpy(sb.tout, sb.pout) for sb in
1667             self.subfields]) / self.massflow, self.pout)
1668
1669 def set_act_tout(self, htf):
1670     """
1671     Calculates HTF output temperature throughout the solar plant as a
1672     weighted average based on the enthalpy of the mass flow in each
1673     loop of the solar field
1674     """
1675     self.act_tout = htf.get_temperature(
1676         np.sum([sb.act_massflow *
1677                 htf.get_enthalpy(sb.act_tout, sb.act_pout) for sb in
1678                     self.subfields]) / self.act_massflow, self.act_pout)
1679
1680 def set_tin(self, htf):

```

```

1681     """
1682     Calculates HTF output temperature throughout the solar plant as a
1683     weighted average based on the enthalpy of the mass flow in each
1684     loop of the solar field
1685     """
1686     self.tin = tf.get_temperature(
1687         np.sum([sb.massflow *
1688                 tf.get_enthalpy(sb.tin, sb.pin) for sb in
1689                 self.subfields]) / self.massflow, self.pin)
1690
1691     def set_pin(self):
1692
1693         self.pin = np.sum([sb.pin * sb.massflow for sb in self.subfields]) \
1694             / self.massflow
1695
1696     def set_act_pin(self):
1697
1698         self.act_pin = np.sum(
1699             [sb.act_pin * sb.act_massflow for sb in self.subfields]) \
1700             / self.massflow
1701
1702     def set_thermal_power(self, htf, datatype):
1703
1704         self.pwr = self.massflow * \
1705             htf.get_delta_enthalpy(self.tin, self.tout, self.pin, self.pout)
1706
1707         # From watts to MW
1708         self.pwr /= 1000000
1709
1710     if datatype == 2:
1711         self.act_pwr = self.act_massflow * \
1712             htf.get_delta_enthalpy(
1713                 self.act_tin, self.act_tout, self.act_pin, self.act_pout)
1714
1715         # From watts to MW
1716         self.act_pwr /= 1000000
1717
1718     def print(self):
1719
1720         for sb in self.subfields:
1721             for l in sb.loops:
1722                 for s in l.scas:
1723                     for h in s.hces:
1724                         print("subfield: ", sb.name,
1725                               "Lazo: ", l.loop_order,
1726                               "SCA: ", s.sca_order,
1727                               "HCE: ", h.hce_order,
1728                               "tin", "=", h.tin,
1729                               "tout", "=", h.tout)
1730
1731
1732 class SolarFieldSimulation(object):
1733     """
1734     Definimos la clase simulacion para representar las diferentes
1735     pruebas que lancemos, variando el archivo TMY, la configuracion del
1736     site, la planta, el modo de operacion o el modelo empleado.
1737     """
1738
1739     def __init__(self, settings):
1740

```

```

1741     self.ID = settings['simulation']['ID']
1742     self.simulation = settings['simulation']['simulation']
1743     self.benchmark = settings['simulation']['benchmark']
1744     self.datatype = settings['simulation']['datatype']
1745     self.fastmode = settings['simulation']['fastmode']
1746     self.tracking = True
1747     self.solarfield = None
1748     self.htf = None
1749     self.coldfluid = None
1750     self.site = None
1751     self.datasource = None
1752     self.powercycle = None
1753     self.parameters = settings
1754     self.first_date = pd.to_datetime(settings['simulation']['first_date'])
1755     self.last_date = pd.to_datetime(settings['simulation']['last_date'])
1756     self.report_df = pd.DataFrame()
1757
1758     if settings['model']['name'] == 'Barbero4thOrder':
1759         self.model = ModelBarbero4thOrder(settings['model'])
1760     elif settings['model']['name'] == 'Barbero1stOrder':
1761         self.model = ModelBarbero1stOrder(settings['model'])
1762     elif settings['model']['name'] == 'BarberoSimplified':
1763         self.model = ModelBarberoSimplified(settings['model'])
1764
1765     if self.datatype == 1:
1766         self.datasource = Weather(settings['simulation'])
1767     elif self.datatype == 2:
1768         self.datasource = FieldData(settings['simulation'],
1769                                     settings['tags'])
1770
1771     if not hasattr(self.datasource, 'site'):
1772         self.site = Site(settings['site'])
1773     else:
1774         self.site = Site(self.datasource.site_to_dict())
1775
1776
1777     if settings['HTF']['source'] == "CoolProp":
1778         if settings['HTF']['CoolPropID'] not in Fluid._COOLPROP_FLUIDS:
1779             print("Not CoolPropID valid")
1780             sys.exit()
1781         else:
1782             self.htf = FluidCoolProp(settings['HTF'])
1783
1784     else:
1785         self.htf = FluidTabular(settings['HTF'])
1786
1787     self.solarfield = SolarField(settings['subfields'],
1788                                  settings['loop'],
1789                                  settings['SCA'],
1790                                  settings['HCE'])
1791
1792     self.base_loop = BaseLoop(settings['loop'],
1793                               settings['SCA'],
1794                               settings['HCE'])
1795
1796     def runSimulation(self):
1797
1798         self.show_message()
1799
1800         for row in self.datasource.dataframe.iterrows():

```

```

1801
1802     if self.datatype == 1: # Because tmy format include TZ info
1803         naive_datetime = datetime.strptime(
1804             row[0].strftime('%Y/%m/%d %H:%M'), "%Y/%m/%d %H:%M")
1805     else:
1806         naive_datetime = row[0]
1807
1808     if (naive_datetime < self.first_date or
1809         naive_datetime > self.last_date):
1810         pass
1811
1812     else:
1813
1814         solarpos = self.site.get_solarposition(row)
1815
1816         self.gather_general_data(row, solarpos)
1817
1818
1819         if solarpos['zenith'][0] < 90:
1820             self.tracking = True
1821         else:
1822             self.tracking = False
1823
1824         if self.simulation:
1825             self.simulate_solarfield(solarpos, row)
1826             self.solarfield.set_thermal_power(self.htf, self.datatype)
1827             self.gather_simulation_data(row)
1828
1829         str_data = ("SIMULATION: {0} " +
1830                     "DNI: {1:3.0f} W/m2 Qm: {2:4.1f}kg/s " +
1831                     "Tin: {3:4.1f}° Tout: {4:4.1f}°C")
1832
1833         print(str_data.format(row[0],
1834                               row[1]['DNI'],
1835                               self.solarfield.massflow,
1836                               self.solarfield.tin - 273.15,
1837                               self.solarfield.tout - 273.15))
1838
1839
1840         if self.benchmark and self.datatype == 2: # 2: Field Data File
available
1841             self.benchmark_solarfield(solarpos, row)
1842             self.solarfield.set_thermal_power(self.htf, self.datatype)
1843             self.gather_benchmark_data(row)
1844
1845             str_data = ("BENCHMARK: {0} " +
1846                         "DNI: {1:3.0f} W/m2 act_Qm: {2:4.1f}kg/s " +
1847                         "act_Tin: {3:4.1f}° act_Tout: {4:4.1f}° " +
1848                         "Tout: {5:4.1f}°")
1849
1850             print(str_data.format(row[0],
1851                                   row[1]['DNI'],
1852                                   self.solarfield.act_massflow,
1853                                   self.solarfield.act_tin - 273.15,
1854                                   self.solarfield.act_tout - 273.15,
1855                                   self.solarfield.tout - 273.15))
1856
1857             self.save_results()
1858
1859     def simulate_solarfield(self, solarpos, row):

```

```

1860
1861     if self.datatype == 1:
1862         for s in self.solarfield.subfields:
1863             s.initialize('rated')
1864             for l in s.loops:
1865                 l.initialize('rated')
1866         self.solarfield.initialize('rated')
1867         self.base_loop.initialize('rated')
1868     if self.fastmode:
1869         if solarpos['zenith'][0] > 90:
1870             self.base_loop.massflow = \
1871                 self.base_loop.parameters['min_massflow']
1872             self.base_loop.calc_loop_pr_for_massflow(
1873                 row, solarpos, self.htf, self.model)
1874         else:
1875             self.base_loop.calc_loop_pr_for_tout(
1876                 row, solarpos, self.htf, self.model)
1877         for s in self.solarfield.subfields:
1878             for l in s.loops:
1879                 l.load_from_base_loop(self.base_loop)
1880     else:
1881         for s in self.solarfield.subfields:
1882             for l in s.loops:
1883                 if solarpos['zenith'][0] > 90:
1884                     l.massflow = \
1885                         self.base_loop.parameters['min_massflow']
1886                     l.calc_loop_pr_for_massflow(
1887                         row, solarpos, self.htf, self.model)
1888                 else:
1889                     if l.loop_order > 1:
1890                         # For a faster convergence
1891                         l.massflow = \
1892                             l.subfield.loops[l.loop_order - 1].massflow
1893                         l.calc_loop_pr_for_tout(
1894                             row, solarpos, self.htf, self.model)
1895
1896     elif self.datatype == 2:
1897         # 1st, we initialize subfields because actual data are given for
1898         # subfields. 2nd, we initialize solarfield.
1899         for s in self.solarfield.subfields:
1900             s.load_actual(row)
1901             s.initialize('actual')
1902         self.solarfield.load_actual(self.htf)
1903         self.solarfield.initialize('actual')
1904     if self.fastmode:
1905         # Force minimum massflow at night
1906         for s in self.solarfield.subfields:
1907             self.base_loop.initialize('subfield', s)
1908             if solarpos['zenith'][0] > 90:
1909                 self.base_loop.massflow = \
1910                     self.base_loop.parameters['min_massflow']
1911                 self.base_loop.calc_loop_pr_for_massflow(
1912                     row, solarpos, self.htf, self.model)
1913             else:
1914                 self.base_loop.calc_loop_pr_for_tout(
1915                     row, solarpos, self.htf, self.model)
1916             for l in s.loops:
1917                 l.load_from_base_loop(self.base_loop)
1918     else:
1919         for s in self.solarfield.subfields:

```

```

1920     for l in s.loops:
1921         # l.load_actual(s)
1922         l.initialize('subfield', s)
1923         if solarpos['zenith'][0] > 90:
1924             l.massflow = l.parameters['min_massflow']
1925             l.calc_loop_pr_for_massflow(
1926                 row, solarpos, self.htf, self.model)
1927         else:
1928             if l.loop_order > 1:
1929                 # For a faster convergence
1930                 l.massflow = \
1931                     l.subfield.loops[l.loop_order - 1].massflow
1932             l.calc_loop_pr_for_tout(
1933                 row, solarpos, self.htf, self.model)
1934
1935     for s in self.solarfield.subfields:
1936         s.set_subfield_values_from_loops(self.htf)
1937
1938     self.solarfield.set_solarfield_values_from_subfields(self.htf)
1939
1940 def benchmark_solarfield(self, solarpos, row):
1941
1942     for s in self.solarfield.subfields:
1943         s.load_actual(row)
1944         s.initialize('actual')
1945     self.solarfield.load_actual(self.htf)
1946     self.solarfield.initialize('actual')
1947
1948     if self.fastmode:
1949         for s in self.solarfield.subfields:
1950             self.base_loop.initialize('subfield', s)
1951             self.base_loop.calc_loop_pr_for_massflow(
1952                 row, solarpos, self.htf, self.model)
1953             self.base_loop.set_loop_values_from_HCEs()
1954
1955         for l in s.loops:
1956             l.load_from_base_loop(self.base_loop)
1957
1958         s.set_subfield_values_from_loops(self.htf)
1959
1960     else:
1961         for s in self.solarfield.subfields:
1962             for l in s.loops:
1963                 # l.load_actual()
1964                 l.initialize('subfield', s)
1965                 l.calc_loop_pr_for_massflow(
1966                     row, solarpos, self.htf, self.model)
1967                 l.set_loop_values_from_HCEs()
1968
1969             s.set_subfield_values_from_loops(self.htf)
1970
1971     self.solarfield.set_solarfield_values_from_subfields(self.htf)
1972
1973
1974 def store_values(self, row, values):
1975
1976     for v in values:
1977         self.datasource.dataframe.at[row[0], v] = values[v]
1978
1979 def gather_general_data(self, row, solarpos):

```

```

1980
1981     self.datasource.dataframe.at[row[0], 'elevation'] = \
1982         solarpos['elevation'][0]
1983     self.datasource.dataframe.at[row[0], 'zenith'] = \
1984         solarpos['zenith'][0]
1985     self.datasource.dataframe.at[row[0], 'azimuth'] = \
1986         solarpos['azimuth'][0]
1987     aoi = self.base_loop.get_aoi(solarpos)
1988     self.datasource.dataframe.at[row[0], 'aoi'] = aoi
1989     self.datasource.dataframe.at[row[0], 'IAM'] = \
1990         self.base_loop.get_IAM(solarpos)
1991     self.datasource.dataframe.at[row[0], 'pr_shadows'] = \
1992         self.base_loop.get_pr_shadows2(solarpos)
1993     self.datasource.dataframe.at[row[0], 'pr_borders'] = \
1994         self.base_loop.get_pr_borders(aoi)
1995     self.datasource.dataframe.at[row[0], 'pr_opt_peak'] = \
1996         self.base_loop.get_pr_opt_peak()
1997     self.datasource.dataframe.at[row[0], 'solar_fraction'] = \
1998         self.base_loop.get_solar_fraction()
1999
2000 def gather_simulation_data(self, row):
2001
2002     # Solarfield data
2003     self.datasource.dataframe.at[row[0], 'SF.x.mf'] = \
2004         self.solarfield.massflow
2005     self.datasource.dataframe.at[row[0], 'SF.x.tin'] = \
2006         self.solarfield.tin - 273.15
2007     self.datasource.dataframe.at[row[0], 'SF.x.tout'] = \
2008         self.solarfield.tout - 273.15
2009     self.datasource.dataframe.at[row[0], 'SF.x.pin'] = \
2010         self.solarfield.pin
2011     self.datasource.dataframe.at[row[0], 'SF.x.pout'] = \
2012         self.solarfield.pout
2013     self.datasource.dataframe.at[row[0], 'SF.x.prth'] = \
2014         self.solarfield.pr
2015     self.datasource.dataframe.at[row[0], 'SF.x.prop'] = \
2016         self.solarfield.pr_opt
2017     self.datasource.dataframe.at[row[0], 'SF.x.qabs'] = \
2018         self.solarfield.qabs
2019     self.datasource.dataframe.at[row[0], 'SF.x.qlst'] = \
2020         self.solarfield.qlost
2021     self.datasource.dataframe.at[row[0], 'SF.x qlbk'] = \
2022         self.solarfield.qlost_brackets
2023     self.datasource.dataframe.at[row[0], 'SF.x.pwr'] = \
2024         self.solarfield.pwr
2025
2026 if self.datatype == 2:
2027     if row[1]['GrossPower']>0:
2028         self.datasource.dataframe.at[row[0], 'SF.x.globalpr'] = \
2029             row[1]['GrossPower'] / self.solarfield.pwr
2030     else:
2031         self.datasource.dataframe.at[row[0], 'SF.x.globalpr'] = 0
2032 else:
2033     self.datasource.dataframe.at[row[0], 'SF.x.globalpr'] = 0
2034
2035 if self.fastmode:
2036
2037     values = {
2038         self.base_loop.get_id() + '.x.mf': self.base_loop.massflow,
2039         self.base_loop.get_id() + '.x.tin':

```

```

2040             self.base_loop.tin - 273.15,
2041             self.base_loop.get_id() + '.x.tout':
2042                 self.base_loop.tout - 273.15,
2043             self.base_loop.get_id() + '.x.pin': self.base_loop.pin,
2044             self.base_loop.get_id() + '.x.pout': self.base_loop.pout,
2045             self.base_loop.get_id() + '.x.prth': self.base_loop.pr,
2046             self.base_loop.get_id() + '.x.prop': self.base_loop.pr_opt,
2047             self.base_loop.get_id() + '.x.qabs': self.base_loop.qabs,
2048             self.base_loop.get_id() + '.x qlst': self.base_loop.qlost,
2049             self.base_loop.get_id() + '.x qlbk': \
2050                 self.base_loop.qlost_brackets}
2051         self.store_values(row, values)
2052
2053     for s in self.solarfield.subfields:
2054         # Agretate data from subfields
2055         values = {
2056             s.get_id() + '.x.mf': s.massflow,
2057             s.get_id() + '.x.tin': s.tin - 273.15,
2058             s.get_id() + '.x.tout': s.tout - 273.15,
2059             s.get_id() + '.x.pin': s.pin,
2060             s.get_id() + '.x.pout': s.pout,
2061             s.get_id() + '.x.prth': s.pr,
2062             s.get_id() + '.x.prop': s.pr_opt,
2063             s.get_id() + '.x.qabs': s.qabs,
2064             s.get_id() + '.x qlst': s.qlost,
2065             s.get_id() + '.x qlbk': s.qlost_brackets}
2066
2067         self.store_values(row, values)
2068
2069     if not self.fastmode:
2070         for l in s.loops:
2071             # Loop data
2072             values = {
2073                 l.get_id() + '.x.mf': l.massflow,
2074                 l.get_id() + '.x.tin': l.tin - 273.15,
2075                 l.get_id() + '.x.tout': l.tout - 273.15,
2076                 l.get_id() + '.x.pin': l.pin,
2077                 l.get_id() + '.x.pout': l.pout,
2078                 l.get_id() + '.x.prth': l.pr,
2079                 l.get_id() + '.x.prop': l.pr_opt,
2080                 l.get_id() + '.x.qabs': l.qabs,
2081                 l.get_id() + '.x.qlost': l.qlost,
2082                 l.get_id() + '.x qlbk': l.qlost_brackets}
2083
2084         self.store_values(row, values)
2085
2086     def gather_benchmark_data(self, row):
2087
2088         # Solarfield data
2089         self.datasource.dataframe.at[row[0], 'SF.a.mf'] = \
2090             self.solarfield.massflow
2091         self.datasource.dataframe.at[row[0], 'SF.a.tin'] = \
2092             self.solarfield.tin - 273.15
2093         self.datasource.dataframe.at[row[0], 'SF.a.tout'] = \
2094             self.solarfield.act_tout - 273.15
2095         self.datasource.dataframe.at[row[0], 'SF.a.pwr'] = \
2096             self.solarfield.act_pwr
2097         self.datasource.dataframe.at[row[0], 'SF.b.tout'] = \
2098             self.solarfield.tout - 273.15
2099         self.datasource.dataframe.at[row[0], 'SF.b.prth'] = \

```

```

2100         self.solarfield.pr
2101     self.datasource.dataframe.at[row[0], 'SF.b.prop'] = \
2102         self.solarfield.pr_opt
2103     self.datasource.dataframe.at[row[0], 'SF.b.pwr'] = \
2104         self.solarfield.pwr
2105     self.datasource.dataframe.at[row[0], 'SF.b.wpwr'] = \
2106         self.solarfield.wasted_power
2107     self.datasource.dataframe.at[row[0], 'SF.a.pin'] = \
2108         self.solarfield.pin
2109     self.datasource.dataframe.at[row[0], 'SF.a.pout'] = \
2110         self.solarfield.act_pout
2111     self.datasource.dataframe.at[row[0], 'SF.b.pout'] = \
2112         self.solarfield.pout
2113     self.datasource.dataframe.at[row[0], 'SF.b.qabs'] = \
2114         self.solarfield.qabs
2115     self.datasource.dataframe.at[row[0], 'SF.b qlst'] = \
2116         self.solarfield.qlost
2117     self.datasource.dataframe.at[row[0], 'SF.b qlbk'] = \
2118         self.solarfield.qlost_brackets
2119
2120     if self.solarfield.qabs > 0:
2121         self.datasource.dataframe.at[row[0], 'SF.a.prth'] = \
2122             self.solarfield.act_pwr / self.solarfield.qabs
2123     else:
2124         self.datasource.dataframe.at[row[0], 'SF.a.prth'] = 0
2125
2126     if row[1]['GrossPower']>0:
2127         self.datasource.dataframe.at[row[0], 'SF.a.globalpr'] = \
2128             row[1]['GrossPower'] / self.solarfield.act_pwr
2129     else:
2130         self.datasource.dataframe.at[row[0], 'SF.a.globalpr'] = 0
2131
2132     if row[1]['GrossPower']>0:
2133         self.datasource.dataframe.at[row[0], 'SF.b.globalpr'] = \
2134             row[1]['GrossPower'] / self.solarfield.pwr
2135     else:
2136         self.datasource.dataframe.at[row[0], 'SF.a.globalpr'] = 0
2137
2138     for s in self.solarfield.subfields:
2139
2140         if self.fastmode:
2141             values = {
2142                 self.base_loop.get_id(s) + '.a.mf':
2143                     self.base_loop.massflow,
2144                 self.base_loop.get_id(s) + '.a.tin':
2145                     self.base_loop.tin - 273.15,
2146                 self.base_loop.get_id(s) + '.a.tout':
2147                     self.base_loop.act_tout - 273.15,
2148                 self.base_loop.get_id(s) + '.b.tout':
2149                     self.base_loop.tout - 273.15,
2150                 self.base_loop.get_id(s) + '.a.pin': self.base_loop.pin,
2151                 self.base_loop.get_id(s) + '.b.pout': self.base_loop.pout,
2152                 self.base_loop.get_id(s) + '.b.prth': self.base_loop.pr,
2153                 self.base_loop.get_id(s) + '.b.prop':
2154                     self.base_loop.pr_opt,
2155                 self.base_loop.get_id(s) + '.b.qabs': self.base_loop.qabs,
2156                 self.base_loop.get_id(s) + '.b qlst': self.base_loop.qlost,
2157                 self.base_loop.get_id(s) + '.b qlbk':
2158                     self.base_loop.qlost_brackets,
2159                 self.base_loop.get_id(s) + '.b wpwr':

```

```

2160             self.base_loop.wasted_power}
2161         self.store_values(row, values)
2162
2163     # Agregate data from subfields
2164     values = {
2165         s.get_id() + '.a.mf': s.act_massflow,
2166         s.get_id() + '.a.tin': s.act_tin - 273.15,
2167         s.get_id() + '.a.tout': s.act_tout - 273.15,
2168         s.get_id() + '.b.tout': s.tout - 273.15,
2169         s.get_id() + '.a.pin': s.act_pin,
2170         s.get_id() + '.a.pout': s.act_pout,
2171         s.get_id() + '.b.pout': s.pout,
2172         s.get_id() + '.b.prth': s.pr,
2173         s.get_id() + '.b.prop': s.pr_opt,
2174         s.get_id() + '.b.qabs': s.qabs,
2175         s.get_id() + '.b.qlost': s.qlost,
2176         s.get_id() + '.b qlbk': s.qlost_brackets,
2177         s.get_id() + '.b.wpwr': s.wasted_power}
2178
2179     self.store_values(row, values)
2180
2181     if not self.fastmode:
2182         for l in s.loops:
2183             # Loop data
2184             values = {
2185                 l.get_id() + '.a.mf': l.act_massflow,
2186                 l.get_id() + '.a.tin': l.act_tin - 273.15,
2187                 l.get_id() + '.a.tout': l.act_tout - 273.15,
2188                 l.get_id() + '.b.tout': l.tout - 273.15,
2189                 l.get_id() + '.a.pin': l.act_pin,
2190                 l.get_id() + '.a.pout': l.act_pout,
2191                 l.get_id() + '.b.pout': l.pout,
2192                 l.get_id() + '.b.prth': l.pr,
2193                 l.get_id() + '.b.prop': l.pr_opt,
2194                 l.get_id() + '.b.qabs': l.qabs,
2195                 l.get_id() + '.b.qlost': l.qlost,
2196                 l.get_id() + '.b qlbk': l.qlost_brackets,
2197                 l.get_id() + '.b.wpwr': l.wasted_power}
2198
2199             self.store_values(row, values)
2200
2201
2202     def show_report(self, keys= None):
2203
2204         self.report_df[keys].plot(
2205             figsize=(20,10), linewidth=5, fontsize=20)
2206         plt.xlabel('Date', fontsize=20)
2207         pd.set_option('display.max_rows', None)
2208         pd.set_option('display.max_columns', None)
2209         pd.set_option('display.width', None)
2210
2211     def save_results(self):
2212
2213         keys = ['DNI', 'elevation', 'zenith', 'azimuth', 'aoi', 'IAM',
2214                'pr_shadows', 'pr_borders', 'pr_opt_peak', 'solar_fraction']
2215
2216         keys_graphics_power = ['DNI']
2217         keys_graphics_temp = ['DNI']
2218
2219         keys_a = ['NetPower', 'AuxPower', 'GrossPower',

```

```

2220             'SF.a.mf', 'SF.a.tin', 'SF.a.tout',
2221             'SF.a.pwr', 'SF.a.prth', 'SF.a.globalpr']
2222     keys_graphics_power_a = ['NetPower', 'AuxPower', 'GrossPower',
2223                               'SF.a.pwr']
2224     keys_graphics_temp_a = ['SF.a.mf', 'SF.a.tin', 'SF.a.tout']
2225
2226     keys_x = ['SF.x.mf', 'SF.x.tin', 'SF.x.tout', 'SF.x.pwr',
2227                'SF.x.prth', 'SF.x.globalpr']
2228     keys_graphics_power_x = ['SF.x.pwr']
2229     keys_graphics_temp_x = ['SF.x.mf', 'SF.x.tout']
2230
2231     keys_b = ['SF.b.tout', 'SF.b.pwr', 'SF.b.wpwr',
2232                'SF.b.prth', 'SF.b.globalpr']
2233     keys_graphics_power_b = ['SF.b.pwr']
2234     keys_graphics_temp_b = ['SF.b.tout']
2235
2236     if self.datatype == 2:
2237         keys += keys_a
2238         keys_graphics_power += keys_graphics_power_a
2239         keys_graphics_temp += keys_graphics_temp_a
2240
2241     if self.simulation == True:
2242         keys += keys_x
2243         keys_graphics_power += keys_graphics_power_x
2244         keys_graphics_temp += keys_graphics_temp_x
2245
2246     if self.benchmark == True:
2247         keys += keys_b
2248         keys_graphics_power += keys_graphics_power_b
2249         keys_graphics_temp += keys_graphics_temp_b
2250
2251     self.report_df = self.datasource.dataframe
2252
2253     try:
2254         initialdir = "./simulations_outputs/"
2255         prefix = datetime.today().strftime("%Y%m%d %H%M%S ")
2256         filename_complete = str(self.ID) + "_COMPLETE"
2257         filename_report = str(self.ID) + "_REPORT"
2258         sufix = ".csv"
2259
2260         path_complete = initialdir + prefix + filename_complete + sufix
2261         path_report = initialdir + prefix + filename_report + sufix
2262
2263         self.datasource.dataframe.to_csv(
2264             path_complete, sep=';', decimal = ',')
2265         # self.report_df.to_csv(path_report, sep=';', decimal = ',')
2266
2267     except Exception:
2268         raise
2269         print('Error saving results, unable to save file: %r', path)
2270
2271     def show_message(self):
2272
2273         print("Running simulation for source data file: {0} from: \
2274             {1} to {2}".format(
2275             self.parameters['simulation']['filename'],
2276             self.parameters['simulation']['first_date'],
2277             self.parameters['simulation']['last_date']))
2278         print("Model: {0}".format(
2279             self.parameters['model']['name']))

```

```

2280     print("Simulation: {0} ; Benchmark: {1} ; FastMode: {2}" .format(
2281         self.parameters['simulation']['simulation'],
2282         self.parameters['simulation']['benchmark'],
2283         self.parameters['simulation']['fastmode']))
2284
2285     print("Site: {0} @ Lat: {1:.2f}º, Long: {2:.2f}º, Alt: {3} m".format(
2286         self.site.name, self.site.latitude,
2287         self.site.longitude, self.site.altitude))
2288
2289     print("Loops:", self.solarfield.total_loops,
2290           'SCA/loop:', self.parameters['loop']['scas'],
2291           'HCE/SCA:', self.parameters['loop']['hces'])
2292     print("SCA model:", self.parameters['SCA']['Name'])
2293     print("HCE model:", self.parameters['HCE']['Name'])
2294     if self.parameters['HTF']['source'] == 'table':
2295         print("HTF form table:", self.parameters['HTF']['name'])
2296     elif self.parameters['HTF']['source'] == 'CoolProp':
2297         print("HTF form CoolProp:", self.parameters['HTF']['CoolPropID'])
2298     print("-----")
2299
2300
2301 class LoopSimulation(object):
2302     """
2303     Definimos la clase simulacion para representar las diferentes
2304     pruebas que lancemos, variando el archivo TMY, la configuracion del
2305     site, la planta, el modo de operacion o el modelo empleado.
2306     """
2307
2308     def __init__(self, settings):
2309
2310         self.tracking = True
2311         self.htf = None
2312         self.site = None
2313         self.datasource = None
2314         self.parameters = settings
2315
2316         if settings['model']['name'] == 'Barbero4thOrder':
2317             self.model = ModelBarbero4thOrder(settings['model'])
2318         elif settings['model']['name'] == 'Barbero1stOrder':
2319             self.model = ModelBarbero1stOrder(settings['model'])
2320         elif settings['model']['name'] == 'BarberoSimplified':
2321             self.model = ModelBarberoSimplified(settings['model'])
2322
2323         self.datasource = TableData(settings['simulation'])
2324
2325         self.site = Site(settings['site'])
2326
2327         if settings['HTF']['source'] == "CoolProp":
2328             if settings['HTF']['CoolPropID'] not in Fluid._COOLPROP_FLUIDS:
2329                 print("Not CoolPropID valid")
2330                 sys.exit()
2331             else:
2332                 self.htf = FluidCoolProp(settings['HTF'])
2333         else:
2334             self.htf = FluidTabular(settings['HTF'])
2335
2336         self.base_loop = BaseLoop(settings['loop'],
2337                               settings['SCA'],
2338                               settings['HCE'])
2339

```

```

2340 def runSimulation(self):
2341     self.show_message()
2342
2343     flag_0 = datetime.now()
2344
2345     for row in self.datasource.dataframe.iterrows():
2346
2347         solarpos = self.site.get_solarposition(row)
2348
2349         if solarpos['zenith'][0] < 90:
2350             self.tracking = True
2351         else:
2352             self.tracking = False
2353
2354         self.simulate_base_loop(solarpos, row)
2355
2356         str_data = ("{} Ang. Zenith: {:.2f} DNI: {} W/m2 " +
2357                     "Qm: {:.1f}kg/s Tin: {:.1f}K Tout: {:.1f}K")
2358
2359         print(str_data.format(row[0], solarpos['zenith'][0],
2360                               row[1]['DNI'], self.base_loop.act_massflow,
2361                               self.base_loop.tin, self.base_loop.tout))
2362
2363     print(self.datasource.dataframe)
2364
2365
2366     flag_1 = datetime.now()
2367     delta_t = flag_1 - flag_0
2368     print("Total runtime: ", delta_t.total_seconds())
2369
2370     self.save_results()
2371
2372 def simulate_base_loop(self, solarpos, row):
2373
2374     values = {'tin': 573,
2375               'pin': 1900000,
2376               'massflow': 4}
2377     self.base_loop.initialize('values', values)
2378     HCE_var = ''
2379     SCA_var = ''
2380
2381     for c in row[1].keys():
2382         if c in self.base_loop.parameters_sca.keys():
2383             SCA_var = c
2384
2385     for c in row[1].keys():
2386         if c in self.base_loop.parameters_hce.keys():
2387             HCE_var = c
2388
2389     for s in self.base_loop.scas:
2390         if SCA_var != '':
2391             s.parameters[SCA_var] = row[1][SCA_var]
2392             aoi = s.get_aoi(solarpos)
2393             for h in s.hces:
2394                 if HCE_var != '':
2395                     h.parameters[HCE_var] = row[1][HCE_var]
2396                     h.set_pr_opt(solarpos)
2397                     h.set_qabs(aoi, solarpos, row)
2398                     h.set_tin()
2399                     h.set_pin()

```

```

2400         h.tout = h.tin
2401         self.model.calc_pr(h, self.htf, row)
2402
2403     self.base_loop.tout = self.base_loop.scas[-1].hces[-1].tout
2404     self.base_loop.pout = self.base_loop.scas[-1].hces[-1].pout
2405     self.base_loop.set_loop_values_from_HCEs('actual')
2406     print('pr', self.base_loop.pr_act_massflow ,
2407           'tout', self.base_loop.tout,
2408           'massflow', self.base_loop.massflow)
2409
2410     if HCE_var + SCA_var != '':
2411         self.datasource.dataframe.at[row[0], HCE_var + SCA_var] = row[1][HCE_var
2412 + SCA_var]
2412         self.datasource.dataframe.at[row[0], 'pr'] = self.base_loop.pr_act_massflow
2413         self.datasource.dataframe.at[row[0], 'tout'] = self.base_loop.tout
2414         self.datasource.dataframe.at[row[0], 'pout'] = self.base_loop.pout
2415         self.datasource.dataframe.at[row[0], 'Z'] = solarpos['zenith'][0]
2416         self.datasource.dataframe.at[row[0], 'E'] = solarpos['elevation'][0]
2417         self.datasource.dataframe.at[row[0], 'aoi'] = aoi
2418
2419 def save_results(self):
2420
2421
2422     try:
2423         initialdir = "./simulations_outputs/"
2424         prefix = datetime.today().strftime("%Y%m%d %H%M%S")
2425         filename = "Loop Simulation"
2426         sufix = ".csv"
2427
2428         path = initialdir + prefix + filename + sufix
2429
2430         self.datasource.dataframe.to_csv(path, sep=';', decimal = ',')
2431
2432     except Exception:
2433         raise
2434         print('Error saving results, unable to save file: %r', path)
2435
2436
2437 def show_message(self):
2438
2439     print("Running simulation for source data file: {0}".format(
2440         self.parameters['simulation']['filename']))
2441
2442     print("Site: {0} @ Lat: {1:.2f}°, Long: {2:.2f}°, Alt: {3} m".format(
2443         self.site.name, self.site.latitude,
2444         self.site.longitude, self.site.altitude))
2445
2446     print('SCA/loop:', self.parameters['loop']['scas'],
2447           'HCE/SCA:', self.parameters['loop']['hces'])
2448     print("SCA model:", self.parameters['SCA']['Name'])
2449     print("HCE model:", self.parameters['HCE']['Name'])
2450     print("HTF:", self.parameters['HTF']['name'])
2451     print("-----")
2452
2453
2454 class Fluid:
2455
2456     _T_REF = 285.856 # Kelvin, T_REF= 12.706 Celsius Degrees
2457     _COOLPROP_FLUIDS = ['Water', 'INCOMP::TVP1', 'INCOMP::S800']
2458

```

```

2459     def test_fluid(self, tmax, tmin, p):
2460
2461         data = []
2462
2463         for tt in range(int(round(tmax)), int(round(tmin)), -5):
2464             data.append({'T': tt,
2465                         'P': p,
2466                         'cp': self.get_specific_heat(tt, p),
2467                         'rho': self.get_density(tt, p),
2468                         'mu': self.get_dynamic_viscosity(tt, p),
2469                         'kt': self.get_thermal_conductivity(tt, p),
2470                         'H': self.get_enthalpy(tt, p),
2471                         'T-H': self.get_temperature(self.get_enthalpy(tt, p), p)})
2472
2473         df = pd.DataFrame(data)
2474         print(round(df, 6))
2475
2476     def get_specific_heat(self, p, t):
2477         pass
2478
2479     def get_density(self, p, t):
2480         pass
2481
2482     def get_thermal_conductivity(self, p, t):
2483         pass
2484
2485     def get_enthalpy(self, p, t):
2486         pass
2487
2488     def get_temperature(self, h, p):
2489         pass
2490
2491     def get_temperature_by_integration(self, tin, q, mf=None, p=None):
2492         pass
2493
2494     def get_dynamic_viscosity(self, t, p):
2495         pass
2496
2497     def get_Reynolds(self, dri, t, p, massflow):
2498
2499         return (4 * massflow /
2500                 (np.pi * dri * self.get_dynamic_viscosity(t,p)))
2501
2502     def get_massflow_from_Reynolds(self, dri, t, p, re):
2503
2504         if t > self.tmax:
2505             t = self.tmax
2506
2507         return re * np.pi * dri * self.get_dynamic_viscosity(t,p) / 4
2508
2509     def get_prandtl(self, t, p):
2510
2511         # Specific heat capacity
2512         cp = self.get_specific_heat(t, p)
2513
2514         # Fluid dynamic viscosity
2515         mu = self.get_dynamic_viscosity(t, p)
2516
2517         # Fluid density
2518         rho = self.get_density(t, p)

```

```

2519     # Fluid thermal conductivity
2520     kf = self.get_thermal_conductivity(t, p)
2521
2522     # Fluid thermal diffusivity
2523     alpha = kf / (rho * cp)
2524
2525     # # Prandtl number
2526     prandtl = cp * mu / kf
2527
2528     return prandtl
2529
2530 class FluidCoolProp(Fluid):
2531
2532     def __init__(self, settings = None):
2533
2534         if settings['source'] == 'table':
2535             self.name = settings['name']
2536             self.tmax = settings['tmax']
2537             self.tmin = settings['tmin']
2538
2539         elif settings['source'] == 'CoolProp':
2540             self.tmax = PropsSI('T_MAX', settings['CoolPropID'])
2541             self.tmin = PropsSI('T_MIN', settings['CoolPropID'])
2542             self.coolpropID = settings['CoolPropID']
2543
2544     def get_density(self, t, p):
2545
2546         if t > self.tmax:
2547             t = self.tmax
2548
2549         return PropsSI('D','T',t,'P', p, self.coolpropID)
2550
2551     def get_dynamic_viscosity(self, t, p):
2552
2553         if t > self.tmax:
2554             t = self.tmax
2555
2556         #p = 1600000
2557         return PropsSI('V','T',t,'P', p, self.coolpropID)
2558
2559     def get_specific_heat(self, t, p):
2560
2561         if t > self.tmax:
2562             t = self.tmax
2563
2564         return PropsSI('C','T',t,'P', p, self.coolpropID)
2565
2566     def get_thermal_conductivity(self, t, p):
2567         ''' Saturated Fluid conductivity at temperature t '''
2568
2569         if t > self.tmax:
2570             t = self.tmax
2571
2572         return PropsSI('L','T',t,'P', p, self.coolpropID)
2573
2574     def get_enthalpy(self, t, p):
2575
2576         if t > self.tmax:
2577             t = self.tmax
2578

```

```

2579     CP.set_reference_state(self.coolpropID, 'ASHRAE')
2580     deltaH = PropSSI('H', 'T', t, 'P', p, self.coolpropID)
2581     CP.set_reference_state(self.coolpropID, 'DEF')
2582
2583     return deltaH
2584
2585 def get_delta_enthalpy(self, t1, t2, p1, p2):
2586
2587     CP.set_reference_state(self.coolpropID, 'ASHRAE')
2588     h1 = PropSSI('H', 'T', t1, 'P', p1, self.coolpropID)
2589     h2 = PropSSI('H', 'T', t2, 'P', p2, self.coolpropID)
2590     CP.set_reference_state(self.coolpropID, 'DEF')
2591
2592     return mf * (h2-h1)
2593
2594 def get_temperature(self, h, p):
2595
2596     CP.set_reference_state(self.coolpropID, 'ASHRAE')
2597     temperature = PropSSI('T', 'H', h, 'P', p, self.coolpropID)
2598     CP.set_reference_state(self.coolpropID, 'DEF')
2599
2600     return temperature
2601
2602 def get_temperature_by_integration(self, t, q, mf = None, p = None):
2603
2604     if t > self.tmax:
2605         t = self.tmax
2606
2607     CP.set_reference_state(self.coolpropID, 'ASHRAE')
2608     hin = PropSSI('H', 'T', t, 'P', p, self.coolpropID)
2609     try:
2610         temperature = PropSSI('T', 'H', hin + q/mf, 'P', p, self.coolpropID)
2611     except:
2612         temperature = self.tmax
2613     CP.set_reference_state(self.coolpropID, 'DEF')
2614
2615     return temperature
2616
2617 class FluidTabular(Fluid):
2618
2619     def __init__(self, settings=None):
2620
2621         self.name = settings['name']
2622         self.cp = settings['cp']
2623         self.rho = settings['rho']
2624         self.mu = settings['mu']
2625         self.kt = settings['kt']
2626         self.h = settings['h']
2627         self.t = settings['t']
2628         self.tmax = settings['tmax']
2629         self.tmin = settings['tmin']
2630
2631
2632     def get_density(self, t, p):
2633
2634         # Dowtherm A.pdf, 2.2 Single Phase Liquid Properties. pg. 8.
2635
2636         poly = np.polynomial.polynomial.Polynomial(self.rho)
2637
2638         return poly(t)

```

```

2639
2640     def get_dynamic_viscosity(self, t, p):
2641
2642         poly = np.polynomial.polynomial.Polynomial(self.mu)
2643
2644         mu = poly(t)
2645
2646         return mu
2647
2648     def get_specific_heat(self, t, p):
2649
2650         poly = np.polynomial.polynomial.Polynomial(self.cp)
2651
2652         return poly(t)
2653
2654     def get_thermal_conductivity(self, t, p):
2655         ''' Saturated Fluid conductivity at temperature t '''
2656
2657         poly = np.polynomial.polynomial.Polynomial(self.kt)
2658
2659         return poly(t)
2660
2661     def get_enthalpy(self, t, p):
2662
2663         poly = np.polynomial.polynomial.Polynomial(self.h)
2664
2665         return poly(t)
2666
2667     def get_delta_enthalpy(self, t1, t2, p1, p2):
2668
2669         cp0, cp1, cp2, cp3, cp4, cp5, cp6, cp7, cp8 = tuple(self.cp)
2670
2671         h = (
2672             (cp0 * t2 + cp1 * t2**2 / 2 + cp2 * t2**3 / 3 +
2673              cp3 * t2**4 / 4 + cp4 * t2**5 / 5 + cp5 * t2**6 / 6 +
2674              cp6 * t2**7 / 7 + cp7 * t2**8 / 8 + cp8 * t2**9 / 9)
2675             -
2676             (cp0 * t1 + cp1 * t1**2 / 2 + cp2 * t1**3 / 3 +
2677              cp3 * t1**4 / 4 + cp4 * t1**5 / 5 + cp5 * t1**6 / 6 +
2678              cp6 * t1**7 / 7 + cp7 * t1**8 / 8 + cp8 * t1**9 / 9))
2679
2680         return h
2681
2682     def get_temperature(self, h, p):
2683
2684         poly = np.polynomial.polynomial.Polynomial(self.t)
2685
2686         return poly(h)
2687
2688     def get_temperature_by_integration(self, tin, h, mf=None, p=None):
2689
2690
2691         cp0, cp1, cp2, cp3, cp4, cp5, cp6, cp7, cp8 = tuple(self.cp)
2692
2693         a0 = (h/mf + cp0 * tin + cp1 * tin**2 / 2 + cp2 * tin**3 / 3 +
2694              cp3 * tin**4 / 4 + cp4 * tin**5 / 5 + cp5 * tin**6 / 6 +
2695              cp6 * tin**7 / 7 + cp7 * tin**8 / 8 + cp8 * tin**9 / 9)
2696
2697         factors = [a0, -cp0, -cp1 / 2, -cp2 / 3, -cp3 / 4, -cp4 / 5, -cp5 / 6,
2698                    -cp6 / 7, -cp7 / 8, -cp8 / 9]

```

```
2699
2700     poly = np.polynomial.polynomial.Polynomial(factors)
2701     roots = poly.roots()
2702
2703     tout_bigger = []
2704     tout_smaller = []
2705
2706     for r in roots:
2707         if r.imag == 0.0:
2708             if r.real >= tin:
2709                 tout_bigger.append(r.real)
2710             else:
2711                 tout_smaller.append(r.real)
2712
2713     if h > 0:
2714         tout = min(tout_bigger)
2715     elif h<0:
2716         tout = max(tout_smaller)
2717     else:
2718         tout = tin
2719
2720
2721
2722 class Weather(object):
2723
2724     def __init__(self, settings = None):
2725
2726         self.dataframe = None
2727         self.site = None
2728         self.weatherdata = None
2729
2730         if settings is not None:
2731             self.openWeatherDataFile(settings['filepath'] +
2732                                     settings['filename'])
2733             # self.file
2734         else:
2735             self.openWeatherDataFile()
2736
2737         self.dataframe = self.weatherdata[0]
2738         self.site = self.weatherdata[1]
2739
2740         self.change_units()
2741         self.filter_columns()
2742
2743
2744     def openWeatherDataFile(self, path = None):
2745
2746         try:
2747             if path is None:
2748                 root = Tk()
2749                 root.withdraw()
2750                 path = askopenfilename(initialdir = ".weather_files/",
2751                                         title = "choose your file",
2752                                         filetypes = (("TMY files","*.tm2"),
2753                                         ("TMY files","*.tm3"),
2754                                         ("csv files","*.csv"),
2755                                         ("all files","*.*")))
2756
2757                 root.update()
2758                 root.destroy()
2759
```

```

2759     if path is None:
2760         return
2761     else:
2762         strfilename, strext = os.path.splitext(path)
2763
2764         if strext == ".csv":
2765             self.weatherdata = pvlib.iotools.tmy.read_tmy3(path)
2766             self.file = path
2767         elif (strext == ".tm2" or strext == ".tmy"):
2768             self.weatherdata = pvlib.iotools.tmy.read_tmy2(path)
2769             self.file = path
2770         elif strext == ".xls":
2771             pass
2772         else:
2773             print("unknow extension ", strext)
2774             return
2775
2776     except Exception:
2777         raise
2778     txMessageBox.showerror('Error loading Weather Data File',
2779                           'Unable to open file: %r', self.file)
2780
2781 def change_units(self):
2782
2783     for c in self.dataframe.columns:
2784         if (c == 'DryBulb') or (c == 'DewPoint'): # From Celsius Degrees to K
2785             self.dataframe[c] *= 0.1
2786             self.dataframe[c] += 273.15
2787         if c=='Pressure': # from mbar to Pa
2788             self.dataframe[c] *= 1e2
2789
2790 def filter_columns(self):
2791
2792     needed_columns = ['DNI', 'DryBulb', 'DewPoint', 'Wspd', 'Pressure']
2793     columns_to_drop = []
2794     for c in self.dataframe.columns:
2795         if c not in needed_columns:
2796             columns_to_drop.append(c)
2797     self.dataframe.drop(columns = columns_to_drop, inplace = True)
2798
2799 def site_to_dict(self):
2800     """
2801     pvlib.iotools realiza modificaciones en los nombres de las columnas.
2802     """
2803
2804     return {"name": 'nombre_site',
2805             "latitude": self.site['latitude'],
2806             "longitude": self.site['longitude'],
2807             "altitude": self.site['altitude']}
2808
2809 class FieldData(object):
2810
2811     def __init__(self, settings, tags = None):
2812         self.filename = settings['filename']
2813         self.filepath = settings['filepath']
2814         self.file = self.filepath + self.filename
2815         self.first_date = pd.to_datetime(settings['first_date'])
2816         self.last_date = pd.to_datetime(settings['last_date'])
2817         self.tags = tags
2818         self.dataframe = None

```

```
2819
2820     self.openFieldDataFile(self.file)
2821     self.rename_columns()
2822     self.change_units()
2823
2824
2825 def openFieldDataFile(self, path = None):
2826     """
2827     fielddata
2828     """
2829
2830
2831     rows_list = []
2832     index_count = 1 # Skip fist row in skiprows: it has got columns names
2833     #dateparse = lambda x: pd.datetime.strptime(x, '%YYYY/%m/%d %H:%M')
2834     try:
2835         if path is None:
2836             root = Tk()
2837             root.withdraw()
2838             path = askopenfilename(initialdir = ".\fielddata_files\",
2839                                     title = "choose your file",
2840                                     filetypes = ((\"csv files\", \"*.csv\"),\n                                     ("all files", \"*.*\")))
2841             root.update()
2842             root.destroy()
2843         if path is None:
2844             return
2845         else:
2846             strfilename, strext = os.path.splitext(path)
2847             if strext == ".csv":
2848                 df = pd.read_csv(
2849                     path, sep=';',
2850                     decimal=',',
2851                     usecols=[0])
2852
2853
2854             df = pd.to_datetime(
2855                 df['date'], format = "%d/%m/%Y %H:%M")
2856
2857             for row in df:
2858                 if (row < self.first_date or
2859                     row > self.last_date):
2860                     rows_list.append(index_count)
2861                 index_count += 1
2862
2863             self.dataframe = pd.read_csv(
2864                 path, sep=';',
2865                 decimal=',',
2866                 dayfirst=True,
2867                 index_col=0,
2868                 skiprows=rows_list)
2869
2870             self.file = path
2871
2872         else:
2873             print("unknow extension ", strext)
2874             return
2875
2876     except Exception:
2877         raise
2878         txMessageBox.showerror('Error loading FieldData File',
```

```

2879             'Unable to open file: %r', self.file)
2880
2881     self.dataframe.index = pd.to_datetime(self.dataframe.index,
2882                                         format= "%d/%m/%Y %H:%M")
2883
2884 def change_units(self):
2885
2886     for c in self.dataframe.columns:
2887         if ('.a.t' in c) or ('DryBulb' in c) or ('Dew' in c):
2888             self.dataframe[c] += 273.15 # From Celsius Degrees to K
2889         if '.a.p' in c:
2890             self.dataframe[c] *= 1e5 # From Bar to Pa
2891         if 'Pressure' in c:
2892             self.dataframe[c] *= 1e2 # From mBar to Pa
2893
2894 def rename_columns(self):
2895
2896     # Replace tags with names as indicated in configuration file
2897     # (field_data_file: tags)
2898
2899     rename_dict = dict(zip(self.tags.values(), self.tags.keys()))
2900     self.dataframe.rename(columns = rename_dict, inplace = True)
2901
2902
2903     # Remove unnecessary columns
2904     columns_to_drop = []
2905     for c in self.dataframe.columns:
2906         if c not in self.tags.keys():
2907             columns_to_drop.append(c)
2908     self.dataframe.drop(columns = columns_to_drop, inplace = True)
2909
2910 class TableData(object):
2911
2912     def __init__(self, settings):
2913         self.filename = settings['filename']
2914         self.filepath = settings['filepath']
2915         self.file = self.filepath + self.filename
2916         self.dataframe = None
2917
2918         self.openDataFile(self.file)
2919
2920
2921     def openDataFile(self, path = None):
2922
2923         ...
2924         Table ddata
2925         ...
2926
2927     try:
2928         if path is None:
2929             root = Tk()
2930             root.withdraw()
2931             path = askopenfilename(initialdir = ".data_files/",
2932                                     title = "choose your file",
2933                                     filetypes = ((("csv files", "*.csv"),
2934                                     ("all files", "*.*"))))
2935             root.update()
2936             root.destroy()
2937         else:
2938             strfilename, strext = os.path.splitext(path)

```

```

2939
2940         if strext == ".csv":
2941             self.dataframe = pd.read_csv(
2942                 path, sep=';',
2943                 decimal=',',
2944                 dayfirst=True,
2945                 index_col=0)
2946
2947             self.file = path
2948         else:
2949             print("unknow extension ", strext)
2950             return
2951
2952             self.dataframe.index = pd.to_datetime(self.dataframe.index,
2953                                         format= "%d/%m/%Y %H:%M")
2954     except Exception:
2955         raise
2956     txMessageBox.showerror('Error loading FieldData File',
2957                           'Unable to open file: %r', self.file)
2958
2959
2960 class Site(object):
2961     def __init__(self, settings):
2962
2963         self.name = settings['name']
2964         self.latitude = settings['latitude']
2965         self.longitude = settings['longitude']
2966         self.altitude = settings['altitude']
2967
2968
2969     def get_solarposition(self, row):
2970
2971         solarpos = pvlib.solarposition.get_solarposition(
2972             row[0] + timedelta(hours=0.5),
2973             self.latitude,
2974             self.longitude,
2975             self.altitude,
2976             pressure=row[1]['Pressure'],
2977             temperature=row[1]['DryBulb'])
2978
2979         return solarpos
2980
2981     def get_hour_angle(self, row, equiation_of_time):
2982
2983         hour_angle = pvlib.solarposition.hour_angle(
2984             row[0] + timedelta(hours=0.5),
2985             self.longitude,
2986             equation_of_time)
2987
2988         return hour_angle
2989
2990 if __name__ == '__main__':
2991
2992     with open("./saved_configurations/TEST_2016_DOWA.json") as simulation_file:
2993         SIMULATION_SETTINGS = json.load(simulation_file)
2994         SIM = cs.SolarFieldSimulation(SIMULATION_SETTINGS)
2995
2996         FLAG_00 = datetime.now()
2997         SIM.runSimulation()
2998         FLAG_01 = datetime.now()

```

```
2999    DELTA_01 = FLAG_01 - FLAG_00
3000    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 CoolProp.CoolProp as CP
15 import pvlib as pvlib
16 import tkinter as tk
17 from tkinter.filedialog import askopenfilename, asksaveasfile
18 import tkinter.ttk as ttk
19 from tkinter import messagebox
20 # recipe-580746-1.py from
21 # http://code.activestate.com/recipes/
22 # 580746-t kinter-treeview-like-a-table-or-multicolumn-listb/
23 import recipe5807461 as table
24 import json
25 from decimal import Decimal
26 from datetime import datetime
27 import pandas as pd
28
29
30 class Interface(object):
31
32     _COOLPROP_FLUIDS = ['Water', 'INCOMP::TVP1', 'INCOMP::S800']
33
34     _MODELS = ['Barbero4thOrder', 'Barbero1stOrder', 'BarberoSimplified']
35
36     _DIR = {
37         'saved_configurations': './saved_configurations/',
38         'site_files': './site_files/',
39         'fluid_files': './fluid_files',
40         'hce_files': './hce_files',
41         'sca_files': './sca_files',
42         'fielddata_files': './fielddata_files',
43         'weather_files': './weather_files'}
44
45     cfg_settings = {
46         'simulation': {},
47         'solar_plant': {},
48         'site': {},
49         'weather': {},
50         'hce': {},
51         'SCA': {},
52         'hot_hce': {},
53         'cold_hce': {},
54         'cycle': {},
55         'hce_model_settings': {},
56         'hce_scattered_params': {},
57         'sca_scattered_params': {}}
58
59     def __init__(self):
60

```

```

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

```

```

121     self.varfastmode.set(False)
122     self.vardatafilename.set('')
123     self.vardatafilepath.set('')
124     self.vardatafileurl.set('')
125     self.varfirstdate.set(pd.to_datetime('2000/01/01 1:00')).strftime(
126         '%Y/%m/%d %H:%M'))
127     self.varlastdate.set(pd.to_datetime('2000/12/31 1:00')).strftime(
128         '%Y/%m/%d %H:%M'))
129     self.cmbmodelname.set('')
130     self.varmodelmaxerrt.set(1.0)
131     self.varmodelmaxerrtro.set(0.1)
132     self.varmodelmaxerrpr.set(0.01)

133
134     self.varsitename.set(0)
135     self.varsitelat.set(0)
136     self.varsitelong.set(0)
137     self.varsitealt.set(0)
138     self.checkoptions()
139     self.checkfastmode()

140
141 # Solarfield configuration
142 self.solarfield_table.table_data = []
143 self.columns_table.table_data = []

144
145     self.vartin.set(0)
146     self.vartout.set(0)
147     self.varpin.set(0)
148     self.varpout.set(0)
149     self.vartmin.set(0)
150     self.vartmax.set(0)
151     self.varratedmassflow.set(0)
152     self.varrecirculation.set(0)
153     self.varscas.set(0)
154     self.varhces.set(0)
155     self.varrowspacing.set(0)
156     self.varscatrackingtype.set(1)
157     self.varfluidtable.set(1)
158     self.varfluidname.set('')
159     self.varfluidtmax.set(0)
160     self.varfluidtmin.set(0)
161     self.fluid_table.table_data = []
162     self.cmbcoolpropID.set('')
163     self.checkfluid()

164
165 # SCA configuration
166     self.varscaname.set('')
167     self.varscalength.set(0)
168     self.varscaperture.set(0)
169     self.varscafocallen.set(0)
170     self.varscIAMF0.set(0)
171     self.varscIAMF1.set(0)
172     self.varscIAMF2.set(0)
173     self.varscareflectance.set(0)
174     self.varscageoaccuracy.set(0)
175     self.varscatracktwist.set(0)
176     self.varscacleanliness.set(0)
177     self.varscafactor.set(0)
178     self.varscavailability.set(0)

179
180 # HCE configuration

```

```

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

```

```

241     self.varsitealt.set(cfg['site']['altitude']))
242     self.checkoptions()
243     self.checkfastmode()
244
245     # Solarfield configuration
246     list_subfields = []
247     for r in cfg['subfields']:
248         list_subfields.append(list(r.values()))
249
250     self.solarfield_table.table_data = list_subfields
251
252     if 'tags' in cfg.keys():
253         list_tags = []
254         for r in cfg['tags']:
255             list_tags.append([r, ' ', cfg['tags'][r]])
256         self.columns_table.table_data = list_tags
257
258         self.vartin.set(cfg['loop']['rated_tin'])
259         self.vartout.set(cfg['loop']['rated_tout'])
260         self.varpin.set(cfg['loop']['rated_pin'])
261         self.varpout.set(cfg['loop']['rated_pout'])
262         self.vartmin.set(cfg['loop']['tmin'])
263         self.vartmax.set(cfg['loop']['tmax'])
264         self.varratedmassflow.set(cfg['loop']['rated_massflow'])
265         self.varrecirculation.set(cfg['loop']['min_massflow'])
266         self.varscas.set(cfg['loop']['scas'])
267         self.varhces.set(cfg['loop']['hces'])
268         self.varrowspacing.set(cfg['loop']['row_spacing'])
269         self.varscatrackingtype.set(cfg['loop']['Tracking Type'])
270
271     # HTF configuration
272     fluid_table = []
273
274     if cfg['HTF']['source'] == 'table':
275         self.varfluidtable.set(1)
276         self.varfluidname.set(cfg['HTF']['name'])
277         self.varfluidtmax.set(cfg['HTF']['tmax'])
278         self.varfluidtmin.set(cfg['HTF']['tmin'])
279         fluid_table.append(['mu'] + cfg['HTF']['mu'])
280         fluid_table.append(['cp'] + cfg['HTF']['cp'])
281         fluid_table.append(['rho'] + cfg['HTF']['rho'])
282         fluid_table.append(['kt'] + cfg['HTF']['kt'])
283         fluid_table.append(['h'] + cfg['HTF']['h'])
284         fluid_table.append(['t'] + cfg['HTF']['t'])
285         self.fluid_table.table_data = fluid_table
286
287     elif cfg['HTF']['source'] == 'CoolProp':
288         self.varfluidtable.set(2)
289         self.cmbcoolpropID.set(cfg['HTF']['CoolPropID'])
290         self.varfluidtmax.set(cfg['HTF']['tmax'])
291         self.varfluidtmin.set(cfg['HTF']['tmin'])
292
293     self.checkfluid()
294
295     # SCA configuration
296     self.varscaname.set(cfg['SCA']['Name'])
297     self.varscalength.set(cfg['SCA']['SCA Length'])
298     self.varscaperture.set(cfg['SCA']['Aperture'])
299     self.varscafocallen.set(cfg['SCA']['Focal Len'])
300     self.varscIAMF0.set(cfg['SCA']['IAM Coefficient F0'])

```

```

301     self.varscaIAMF1.set(cfg['SCA']['IAM Coefficient F1'])
302     self.varscaIAMF2.set(cfg['SCA']['IAM Coefficient F2'])
303     self.varscareflectance.set(cfg['SCA']['Reflectance'])
304     self.varscageoaccuracy.set(cfg['SCA']['Geom.Accuracy'])
305     self.varscatracktwist.set(cfg['SCA']['Track Twist'])
306     self.varscacleanliness.set(cfg['SCA']['Cleanliness'])
307     self.varscafactor.set(cfg['SCA']['Factor'])
308     self.varscavailability.set(cfg['SCA']['Availability'])

309
310     # HCE configuration
311     self.varhcename.set(cfg['HCE']['Name'])
312     self.varhcedri.set(cfg['HCE']['Absorber tube inner diameter'])
313     self.varhcedro.set(cfg['HCE']['Absorber tube outer diameter'])
314     self.varhcedgi.set(cfg['HCE']['Glass envelope inner diameter'])
315     self.varhcedgo.set(cfg['HCE']['Glass envelope outer diameter'])
316     self.varhcelength.set(cfg['HCE']['Length'])
317     self.varbellowsratio.set(cfg['HCE']['Bellows ratio'])
318     self.varshieldshading.set(cfg['HCE']['Shield shading'])
319     self.varhceemittanceA0.set(cfg['HCE']['Absorber emittance factor A0'])
320     self.varhceemittanceA1.set(cfg['HCE']['Absorber emittance factor A1'])
321     self.varhceabsorptance.set(cfg['HCE']['Absorber absorptance'])
322     self.varhcetransmittance.set(cfg['HCE']['Envelope transmittance'])
323     self.varhceinnerroughness.set(cfg['HCE']['Inner surface roughness'])
324     self.varhceminreynolds.set(cfg['HCE']['Min Reynolds'])
325     self.varhcebrackets.set(cfg['HCE']['Brackets'])
326     self.updateHCEperSCA()

327
328     self.fr_fluid.update()
329     self.fr_hce.update()
330     self.fr_solarfield.update()
331     self.fr_sca.update()
332     self.fr_simulation.update()

333
334     def simulation_save(self):
335
336         self.save_as_JSON(self.generate_json(), self.filename)

337
338
339     def simulation_save_as(self):
340
341         self.save_as_JSON(self.generate_json())

342
343     def generate_json(self):
344
345         self.tagslist = []

346
347         cfg = dict({'simulation': {},
348                     'model': {},
349                     'site': {},
350                     'loop': {},
351                     'SCA': {},
352                     'HCE': {},
353                     'HTF': {}})
354
355
356         cfg['simulation']['ID'] = self.varsimID.get()
357         cfg['simulation']['datatype'] = self.varsimdatatype.get()
358         cfg['simulation']['simulation'] = self.varsimulation.get()
359         cfg['simulation']['benchmark'] = self.varbenchmark.get()
360         cfg['simulation']['fastmode'] = self.varfastmode.get()

```

```

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

```

```

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

```

```

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

```

```

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

```

```
601     var = self.varfastmode.get()
602
603     if var:
604         self.varfastmodetext.set('ON')
605     else:
606         self.varfastmodetext.set('OFF')
607
608 def buildSimulationFrame(self):
609
610     self.varsimID = tk.StringVar(self.fr_simulation)
611     self.varsimdatatype = tk.IntVar(self.fr_simulation)
612     self.tagslist = []
613     self.varsimulation = tk.BooleanVar(self.fr_simulation)
614     self.varbenchmark = tk.BooleanVar(self.fr_simulation)
615     self.varfastmode = tk.BooleanVar(self.fr_simulation)
616     self.varfastmodetext = tk.StringVar(self.fr_simulation)
617     self.varfirstdate = tk.StringVar(self.fr_simulation)
618     self.varlastdate = tk.StringVar(self.fr_simulation)
619
620     self.varmodelmaxerrtro = tk.DoubleVar(self.fr_simulation)
621     self.varmodelmaxerrrt = tk.DoubleVar(self.fr_simulation)
622     self.varmodelmaxerrpr = tk.DoubleVar(self.fr_simulation)
623
624     self.varfastmode.set(False)
625     self.checkfastmode()
626
627     self.varloadsitedata = tk.BooleanVar(self.fr_simulation)
628     self.varloadsitedata.set(False)
629
630     self.varsitename = tk.StringVar(self.fr_simulation)
631     self.varsitelat = tk.DoubleVar(self.fr_simulation)
632     self.varsiteelong = tk.DoubleVar(self.fr_simulation)
633     self.varsitealt = tk.DoubleVar(self.fr_simulation)
634
635     self.vardatafileurl = tk.StringVar(self.fr_simulation)
636     self.vardatafilename = tk.StringVar(self.fr_simulation)
637     self.vardatafilepath = tk.StringVar(self.fr_simulation)
638
639     self.lbsimID = ttk.Label(
640         self.fr_simulation,
641         text='Simulation ID').grid(
642             row=0, column=0, sticky='W', padx=2, pady=5)
643     self.ensimID = ttk.Entry(
644         self.fr_simulation,
645         textvariable=self.varsimID).grid(
646             row=0, column=1, sticky='W', padx=2, pady=5)
647
648     self.lbmodelname = ttk.Label(
649         self.fr_simulation,
650         text='Model name').grid(
651             row=0, column=2, sticky='E', padx=2, pady=5)
652
653     self.cmbmodelname = ttk.Combobox(self.fr_simulation)
654     self.cmbmodelname['values'] = self._MODELS
655     self.cmbmodelname['state'] = 'readonly'
656     self.cmbmodelname.current(0)
657     self.cmbmodelname.grid(row=0, column=3, sticky='W', padx=2, pady=5)
658
659     self.frame2= ttk.Frame(self.fr_simulation)
660     self.frame2.grid(row=1, column=0, columnspan=6, padx=2, pady=5)
```

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

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

1077     path = askopenfilename(
1078         initialdir=self._DIR['fielddata_files'],
1079         title='choose your file',
1080         filetypes=(( 'csv files', '*.csv'),
1081                     ('all files', '*.*')))
1082
1083     self.vardatafilepath.set(os.path.dirname(path)+'/')
1084     self.vardatafilename.set(os.path.basename(path))
1085     self.vardatafileurl.set(os.path.dirname(path)+'/ ' +
1086                             os.path.basename(path))
1087
1088 else:
1089     tk.messagebox.showwarning(
1090         title='Warning',
1091         message='Check Data Source Selection')
1092
1093 if self.varloadsitedata.get():
1094
1095     strfilename, strext = os.path.splitext(path)
1096     if strext == '.csv':
1097         weatherdata = pvlib.iotools.tmy.read_tmy3(path)
1098         file = path
1099     elif (strext == '.tm2' or strext == '.tmy'):
1100         weatherdata = pvlib.iotools.tmy.read_tmy2(path)
1101         file = path
1102     elif strext == '.xls':
1103         pass
1104     else:
1105         print('unknow extension ', strext)
1106         return
1107
1108     self.varsitename.set(weatherdata[1]['City'])
1109     self.varsitelat.set(weatherdata[1]['latitude'])
1110     self.varsitelong.set(weatherdata[1]['longitude'])
1111     self.varsitealt.set(weatherdata[1]['altitude'])
1112
1113 def select_fluid_from_csv(self):
1114
1115     # abre una ventana de selección de csv.
1116     self.loadWindow= tk.Tk()
1117     self.loadWindow.attributes('-fullscreen', False)
1118     self.loadWindow.title('Selection Window')
1119
1120     # Menu
1121     self.loadWindow.menubar = tk.Menu(
1122         self.loadWindow)
1123     self.loadWindow.load_menu = tk.Menu(
1124         self.loadWindow.menubar, tearoff=0)
1125     self.loadWindow.load_menu.add_command(
1126         label='Open', command=self.open_csv())
1127     self.loadWindow.load_menu.add_separator()
1128     self.loadWindow.load_menu.add_command(
1129         label='Exit', command=None)
1130     self.loadWindow.menubar.add_cascade(
1131         label='Select File', menu=self.loadWindow.load_menu)
1132
1133     self.loadWindow.config(menu=self.loadWindow.menubar)
1134
1135 def open_csv(self, path=None):
1136

```

```

1137     df = pd.DataFrame()
1138
1139     try:
1140         if path is None:
1141             root = Tk()
1142             root.withdraw()
1143             path = askopenfilename(initialdir = '.fielddata_files/',
1144                                     title='choose your file',
1145                                     filetypes=([('csv files','*.csv'),
1146                                     ('all files','*.*')))
1147             root.update()
1148             root.destroy()
1149
1150         if path is None:
1151             return
1152         else:
1153             strfilename, strext = os.path.splitext(path)
1154
1155         if strext == '.csv':
1156             print('csv.....')
1157
1158             df = pd.read_csv(path, sep=';',
1159                               decimal= ',')
1159             file = path
1160         else:
1161             print('unknow extension ', strext)
1162             return
1163
1164     else:
1165         strfilename, strext = os.path.splitext(path)
1166
1167     if strext == '.csv':
1168         print('csv...')
1169         df = pd.read_csv(path, sep=';',
1170                           decimal= ',')
1170         file = path
1171     else:
1172         print('unknow extension ', strext)
1173         return
1174
1175
1176     except Exception:
1177         raise
1178
1179 def load_fluid_library(self):
1180
1181     path = askopenfilename(initialdir = self._DIR['fluid_files'],
1182                           title='choose your file',
1183                           filetypes=[('JSON files', '*.json')])
1184
1185     with open(path) as cfg_file:
1186         cfg = json.load(cfg_file, parse_float= float, parse_int= int)
1187
1188     self.fluid_list = cfg
1189
1190     self.cmbfluidname['values'] = [s['name'] for s in cfg ]
1191     self.cmbfluidname.current(0)
1192     self.load_fluid_parameters()
1193
1194 def load_fluid_parameters(self):
1195
1196     self.fluid_config = {}

```

```

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

```

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

```

```

1317         self.varcoolproptmax.set(CP.PropsSI("TMAX", self.cmbcoolpropID.get()))
1318
1319
1320     def buildFluidFrame(self):
1321
1322         self.varfluidtmin = tk.DoubleVar(self.fr_fluid)
1323         self.varfluidtmax = tk.DoubleVar(self.fr_fluid)
1324         self.varcoolproptmin = tk.DoubleVar(self.fr_fluid)
1325         self.varcoolproptmax = tk.DoubleVar(self.fr_fluid)
1326         self.varfluidname = tk.StringVar(self.fr_fluid)
1327         self.fluid_list = []
1328         self.varfluidtable = tk.IntVar(self.fr_fluid)
1329
1330         self.varfluidtable.set(1) # 1 for Table, 2 for CoolProp Library
1331
1332         # RadioButton for fluid from library
1333         self.rbfliudlib = tk.Radiobutton(
1334             self.fr_fluid,
1335             padx=1,
1336             text='Fluid Data from CoolProp',
1337             variable=self.varfluidtable, value=2,
1338             command=lambda: self.checkfluid()).grid(
1339                 row=0, column=0, sticky='W', padx=1, pady=5)
1340
1341         self.lbcoolpropID = tk.Label(
1342             self.fr_fluid, text='CoolProp ID (INCOMP::xxxx)')
1343         self.lbcoolpropID.grid(row=0, column=1, sticky='W', padx=1, pady=5)
1344
1345         self.cmbcoolpropID = ttk.Combobox(self.fr_fluid)
1346         self.cmbcoolpropID.bind(
1347             "<>ComboboxSelected>", lambda event: self.get_coolprop_data())
1348         self.cmbcoolpropID['values'] = self._COOLPROP_FLUIDS
1349         self.cmbcoolpropID['state'] = 'readonly'
1350         self.cmbcoolpropID.current(0)
1351         self.cmbcoolpropID.grid(row=0, column=2, sticky='W', padx=1, pady=5)
1352
1353         self.lbcoolproptmintext = tk.Label(
1354             self.fr_fluid,
1355             text='Tmin[K]')
1356         self.lbcoolproptmintext.grid(
1357             row=0, column=3, sticky='E', padx=1, pady=5)
1358
1359         self.lbcoolproptmin = tk.Label(
1360             self.fr_fluid,
1361             textvariable=self.varcoolproptmin)
1362         self.lbcoolproptmin.grid(
1363             row=0, column=4, sticky='E', padx=1, pady=5)
1364
1365         self.lbcoolproptmaxtext = tk.Label(
1366             self.fr_fluid,
1367             text='Tmax[K]')
1368         self.lbcoolproptmaxtext.grid(
1369             row=0, column=5, sticky='W', padx=1, pady=5)
1370
1371         self.lbcoolproptmax = tk.Label(
1372             self.fr_fluid,
1373             textvariable=self.varcoolproptmax)
1374         self.lbcoolproptmax.grid(
1375             row=0, column=6, sticky='W', padx=1, pady=5)
1376

```

```

1377
1378     self.separator = ttk.Separator(self.fr_fluid).grid(
1379         row=1, column=0, columnspan=99, sticky=(tk.W, tk.E))
1380
1381     # RadioButton for fluid from table
1382     self.rbfluidtable = tk.Radiobutton(
1383         self.fr_fluid,
1384         padx=2,
1385         text = 'Fluid Data from table',
1386         variable=self.varfluidtable, value=1,
1387         command=lambda: self.checkfluid()).grid(
1388             row=2, column=0, columnspan=2, sticky='W', padx=2, pady=5)
1389
1390     self.lbfluidname = ttk.Label(self.fr_fluid, text='Name')
1391     self.lbfluidname.grid(row=3, column=0, sticky='W', padx=2, pady=5)
1392     self.enfluidname = ttk.Entry(self.fr_fluid, textvariable=self.varfluidname)
1393     self.enfluidname.grid(row=3, column=1, sticky='W', padx=2, pady=5)
1394
1395     self.cmbfluidname = ttk.Combobox(self.fr_fluid)
1396     self.cmbfluidname.bind('<<ComboboxSelected>>',
1397                             lambda event: self.load_fluid_parameters())
1398     self.cmbfluidname['values'] = self.fluid_list
1399     self.cmbfluidname.grid(row=3, column=2, padx=5, pady=5)
1400
1401     self.btloadfluidcfg = ttk.Button(
1402         self.fr_fluid, text='Load config',
1403         command=lambda: self.load_fluid_library())
1404     self.btloadfluidcfg.grid(
1405         row=3, column=3, sticky='W', padx=2, pady=5)
1406
1407     self.lbfluidtmin = ttk.Label(self.fr_fluid, text='Tmin[K]')
1408     self.lbfluidtmin.grid(row=4, column=0, sticky='W', padx=2, pady=5)
1409     self.enfluidtmin = ttk.Entry(
1410         self.fr_fluid, textvariable=self.varfluidtmin)
1411     self.enfluidtmin.grid(row=4, column=1, sticky='W', padx=2, pady=5)
1412
1413     self.lbfluidtmax = ttk.Label(self.fr_fluid, text='Tmax [K]')
1414     self.lbfluidtmax.grid(row=4, column=2, sticky='W', padx=2, pady=5)
1415     self.enfluidtmax = ttk.Entry(self.fr_fluid, textvariable=self.varfluidtmax)
1416     self.enfluidtmax.grid(row=4, column=3, sticky='W', padx=2, pady=5)
1417
1418     self.fluid_table = table.Tk_Table(
1419         self.fr_fluid,
1420         ['Parameter',
1421          'A x^0', 'B x^1',
1422          'C x^2', 'D x^3',
1423          'E x^4', 'F x^5',
1424          'G x^6', 'H x^7',
1425          'I x^8'],
1426         row_numbers=True,
1427         stripped_rows=('white', '#f2f2f2'),
1428         select_mode='none',
1429         cell_anchor='e',
1430         adjust_heading_to_content=True)
1431     self.fluid_table.grid(
1432         row=5, column=0, columnspan=7, sticky='W', padx=2, pady=5)
1433
1434     self.checkfluid()
1435
1436 # SCA Construction tab

```

```

1437     def load_sca_library(self):
1438
1439         path = askopenfilename(initialdir=self._DIR['sca_files'],
1440                               title='choose your file',
1441                               filetypes=[('JSON files', '*.json')])
1442
1443         with open(path) as cfg_file:
1444             cfg = json.load(cfg_file, parse_float= float, parse_int= int)
1445
1446             self.sca_list = cfg
1447             self.cmbscaname[ 'values' ] = [s[ 'Name' ] for s in cfg ]
1448             self.cmbscaname.current(0)
1449             self.load_sca_parameters()
1450             self.checkfluid()
1451
1452
1453     def load_sca_parameters(self):
1454
1455         self.sca_config = {}
1456
1457         for item in self.sca_list:
1458             if item[ 'Name' ] == self.cmbscaname.get():
1459                 self.sca_config = item
1460
1461             self.varscname.set(self.sca_config[ 'Name' ])
1462             self.varscalength.set(self.sca_config[ 'SCA Length' ])
1463             self.varscaaperture.set(self.sca_config[ 'Aperture' ])
1464             self.varscafocallen.set(self.sca_config[ 'Focal Len' ])
1465             self.varscIAMF0.set(self.sca_config[ 'IAM Coefficient F0' ])
1466             self.varscIAMF1.set(self.sca_config[ 'IAM Coefficient F1' ])
1467             self.varscIAMF2.set(self.sca_config[ 'IAM Coefficient F2' ])
1468             self.varscareflectance.set(self.sca_config[ 'Reflectance' ])
1469             self.varscageoaccuracy.set(self.sca_config[ 'Geom.Accuracy' ])
1470             self.varscatracktwist.set(self.sca_config[ 'Track Twist' ])
1471             self.varscacleanliness.set(self.sca_config[ 'Cleanliness' ])
1472             self.varscafactor.set(self.sca_config[ 'Factor' ])
1473             self.varscavailability.set(self.sca_config[ 'Availability' ])
1474             self.updateHCEperSCA()
1475
1476     def buildSCAFrame(self):
1477
1478         self.sca_list = []
1479
1480         self.varscname = tk.StringVar(self.fr_sca)
1481         self.varscalength = tk.DoubleVar(self.fr_sca)
1482         self.varscaaperture = tk.DoubleVar(self.fr_sca)
1483         self.varscafocallen = tk.DoubleVar(self.fr_sca)
1484         self.varscIAMF0 = tk.DoubleVar(self.fr_sca)
1485         self.varscIAMF1 = tk.DoubleVar(self.fr_sca)
1486         self.varscIAMF2 = tk.DoubleVar(self.fr_sca)
1487         self.varscareflectance = tk.DoubleVar(self.fr_sca)
1488         self.varscageoaccuracy = tk.DoubleVar(self.fr_sca)
1489         self.varscatracktwist = tk.DoubleVar(self.fr_sca)
1490         self.varscacleanliness = tk.DoubleVar(self.fr_sca)
1491         self.varscafactor = tk.DoubleVar(self.fr_sca)
1492         self.varscavailability = tk.DoubleVar(self.fr_sca)
1493
1494         self.lbscname = ttk.Label(
1495             self.fr_sca,
1496             text='SCA base name').grid(

```

```
1497             row=0, column=0, sticky='W', padx=2, pady=5)
1498
1499     self.enscaname = ttk.Entry(
1500         self.fr_sca,
1501         textvariable=self.vars cname).grid(
1502             row=0, column=1, sticky='W', padx=2, pady=5)
1503
1504     self.btscaload = ttk.Button(
1505         self.fr_sca, text='Load Library',
1506         command=lambda: self.load_sca_library())
1507     self.btscaload.grid(row=0, column=7, sticky='W', padx=2, pady=5)
1508
1509     self.cmbscaname = ttk.Combobox(self.fr_sca)
1510     self.cmbscaname.bind(
1511         '<<ComboboxSelected>>', lambda event: self.load_sca_parameters())
1512     self.cmbscaname['values'] = self.sca_list
1513     self.cmbscaname.grid(row=0, column=3, columnspan=4, padx=5, pady=5)
1514
1515     self.lbscalength = ttk.Label(
1516         self.fr_sca,
1517         text='SCA Length [m]').grid(
1518             row=1, column=0, sticky='W', padx=2, pady=5)
1519     self.enscalength = ttk.Entry(
1520         self.fr_sca,
1521         textvariable=self.varscalength)
1522     self.enscalength.bind('<Key>', lambda event: self.updateHCEperSCA())
1523     self.enscalength.grid(
1524             row=1, column=1, sticky='W', padx=2, pady=5)
1525
1526     self.lbscaaperture = ttk.Label(
1527         self.fr_sca,
1528         text='SCA aperture [m]').grid(
1529             row=2, column=0, sticky='W', padx=2, pady=5)
1530     self.enscaaperture = ttk.Entry(
1531         self.fr_sca,
1532         textvariable=self.varscaperture).grid(
1533             row=2, column=1, sticky='W', padx=2, pady=5)
1534
1535     self.lbscafocallen = ttk.Label(
1536         self.fr_sca,
1537         text='SCA focal length [m]').grid(
1538             row=3, column=0, sticky='W', padx=2, pady=5)
1539     self.enscafocallen = ttk.Entry(
1540         self.fr_sca,
1541         textvariable=self.varscafocallen).grid(
1542             row=3, column=1, sticky='W', padx=2, pady=5)
1543
1544     self.lbscaIAMF0 = ttk.Label(
1545         self.fr_sca,
1546         text='SCA IAM facotr F0 []').grid(
1547             row=4, column=0, sticky='W', padx=2, pady=5)
1548     self.enscaIAMF0 = ttk.Entry(
1549         self.fr_sca,
1550         textvariable=self.varsc IAMF0).grid(
1551             row=4, column=1, sticky='W', padx=2, pady=5)
1552
1553     self.lbscaIAMF1 = ttk.Label(
1554         self.fr_sca,
1555         text='SCA IAM facotr F1 []').grid(
1556             row=5, column=0, sticky='W', padx=2, pady=5)
```

```
1557     self.enscaIAMF1 = ttk.Entry(
1558         self.fr_sca,
1559         textvariable=self.varscaIAMF1).grid(
1560             row=5, column=1, sticky='W', padx=2, pady=5)
1561
1562     self.lbscaIAMF2 = ttk.Label(
1563         self.fr_sca,
1564         text='SCA IAM facotr F2 []').grid(
1565             row=6, column=0, sticky='W', padx=2, pady=5)
1566     self.enscaIAMF2 = ttk.Entry(
1567         self.fr_sca,
1568         textvariable=self.varscaIAMF2).grid(
1569             row=6, column=1, sticky='W', padx=2, pady=5)
1570
1571     self.lbscareflectance = ttk.Label(
1572         self.fr_sca,
1573         text='SCA mirror reflectance []').grid(
1574             row=7, column=0, sticky='W', padx=2, pady=5)
1575     self.enscareflectance = ttk.Entry(
1576         self.fr_sca,
1577         textvariable=self.varscareflectance).grid(
1578             row=7, column=1, sticky='W', padx=2, pady=5)
1579
1580     self.lbscageoaccuracy = ttk.Label(
1581         self.fr_sca,
1582         text='SCA geometric accuracy []').grid(
1583             row=8, column=0, sticky='W', padx=2, pady=5)
1584     self.enscageoaccuracy = ttk.Entry(
1585         self.fr_sca,
1586         textvariable=self.varscageoaccuracy).grid(
1587             row=8, column=1, sticky='W', padx=2, pady=5)
1588
1589     self.lbscatracketwist = ttk.Label(
1590         self.fr_sca,
1591         text='SCA Tracking Twist []').grid(
1592             row=9, column=0, sticky='W', padx=2, pady=5)
1593     self.enscatracketwist = ttk.Entry(
1594         self.fr_sca,
1595         textvariable=self.varscatracketwist).grid(
1596             row=9, column=1, sticky='W', padx=2, pady=5)
1597
1598     self.lbscacleanliness = ttk.Label(
1599         self.fr_sca,
1600         text='SCA Cleanliness []').grid(
1601             row=10, column=0, sticky='W', padx=2, pady=5)
1602     self.enscacleanliness = ttk.Entry(
1603         self.fr_sca,
1604         textvariable=self.varscacleanliness).grid(
1605             row=10, column=1, sticky='W', padx=2, pady=5)
1606
1607     self.lbscafactor = ttk.Label(
1608         self.fr_sca,
1609         text='SCA Factor []').grid(
1610             row=11, column=0, sticky='W', padx=2, pady=5)
1611     self.enscafactor = ttk.Entry(
1612         self.fr_sca,
1613         textvariable=self.varscafactor).grid(
1614             row=11, column=1, sticky='W', padx=2, pady=5)
1615
1616     self.lbscaavailability = ttk.Label(
```

```

1617     self.fr_sca,
1618     text='SCA Availability []').grid(
1619         row=12, column=0, sticky='W', padx=2, pady=5)
1620 self.ensaavailability = ttk.Entry(
1621     self.fr_sca,
1622     textvariable=self.varscaavailability).grid(
1623         row=12, column=1, sticky='W', padx=2, pady=5)
1624
1625 def load_hce_parameters(self):
1626
1627     self.hce_config = {}
1628
1629     for item in self.hce_list:
1630
1631         if item['Name'] == self.cmbhcename.get():
1632             self.hce_config = item
1633
1634         self.varhcename.set(self.hce_config['Name'])
1635         self.varhcedri.set(self.hce_config['Absorber tube inner diameter'])
1636         self.varhcedro.set(self.hce_config['Absorber tube outer diameter'])
1637         self.varhcedgi.set(self.hce_config['Glass envelope inner diameter'])
1638         self.varhcedgo.set(self.hce_config['Glass envelope outer diameter'])
1639         self.varhcelength.set(self.hce_config['Length'])
1640         self.varhceinnerroughness.set(self.hce_config['Inner surface roughness'])
1641         self.varhceminreynolds.set(self.hce_config['Min Reynolds'])
1642         self.varhceemittanceA0.set(self.hce_config['Absorber emittance factor A0'])
1643         self.varhceemittanceA1.set(self.hce_config['Absorber emittance factor A1'])
1644         self.varhceabsorptance.set(self.hce_config['Absorber absorptance'])
1645         self.varhcetransmittance.set(self.hce_config['Envelope transmittance'])
1646         self.varhcebrackets.set(self.hce_config['Brackets'])
1647         self.updateHCEperSCA()
1648
1649 def load_hce_library(self, path = None):
1650
1651     if path is None:
1652         path = askopenfilename(initialdir = self._DIR['hce_files'],
1653                               title='choose your file',
1654                               filetypes=[('JSON files', '*.json')])
1655
1656     with open(path) as cfg_file:
1657         cfg = json.load(cfg_file, parse_float= float, parse_int= int)
1658
1659     self.hce_list = cfg
1660     self.cmbhcename['values'] = [s['Name'] for s in cfg]
1661     self.cmbhcename.current(0)
1662     self.load_hce_parameters()
1663
1664
1665 def updateHCEperSCA(self):
1666
1667     var1 = self.varscalength.get()
1668     var2 = self.varhcelength.get()
1669
1670     if var2 != 0:
1671         self.varhcepersca.set(var1 // var2)
1672     else:
1673         self.varhcepersca.set(0.0)
1674
1675     var3 = self.varhcepersca.get()
1676     var4 = self.varhces.get()

```

```

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

```

```

1736         text='HCE outer diameter [m]').grid(
1737             row=2, column=0, sticky='W', padx=2, pady=5)
1738 self.enhcedro = ttk.Entry(
1739     self.fr_hce,
1740     textvariable=self.varhcedro).grid(
1741         row=2, column=1, sticky='W', padx=2, pady=5)
1742
1743 self.lbhcedgi = ttk.Label(
1744     self.fr_hce,
1745     text='HCE glass inner diameter [m]').grid(
1746         row=3, column=0, sticky='W', padx=2, pady=5)
1747 self.enhcedgi = ttk.Entry(
1748     self.fr_hce,
1749     textvariable=self.varhcedgi).grid(
1750         row=3, column=1, sticky='W', padx=2, pady=5)
1751
1752 self.lbhcedgo = ttk.Label(
1753     self.fr_hce,
1754     text='HCE glass outer diameter [m]').grid(
1755         row=4, column=0, sticky='W', padx=2, pady=5)
1756 self.enhcedgo = ttk.Entry(
1757     self.fr_hce,
1758     textvariable=self.varhcedgo).grid(
1759         row=4, column=1, sticky='W', padx=2, pady=5)
1760
1761 self.lbhcelong = ttk.Label(
1762     self.fr_hce,
1763     text='HCE longitude [m]').grid(
1764         row=5, column=0, sticky='W', padx=2, pady=5)
1765 self.enhcelong = ttk.Entry(
1766     self.fr_hce,
1767     textvariable=self.varhcelength)
1768 self.enhcelong.bind('<Key>', lambda event: self.updateHCEperSCA())
1769 self.enhcelong.grid(row=5, column=1, sticky='W', padx=2, pady=5)
1770
1771 self.lbhcepersca = ttk.Label(
1772     self.fr_hce,
1773     textvariable=self.varhceperscatext).grid(
1774         row=5, column=2, sticky='W', padx=2, pady=5)
1775
1776 self.lbbellowsratio = ttk.Label(
1777     self.fr_hce,
1778     text='Bellows ratio [ ]').grid(
1779         row=6, column=0, sticky='W', padx=2, pady=5)
1780 self.enbellowsratio = ttk.Entry(
1781     self.fr_hce,
1782     textvariable=self.varbellowsratio).grid(
1783         row=6, column=1, sticky='W', padx=2, pady=5)
1784
1785 self.lbshieldshading = ttk.Label(
1786     self.fr_hce,
1787     text='Shield shading [ ]').grid(
1788         row=7, column=0, sticky='W', padx=2, pady=5)
1789 self.enshieldshading = ttk.Entry(
1790     self.fr_hce,
1791     textvariable=self.varshieldshading).grid(
1792         row=7, column=1, sticky='W', padx=2, pady=5)
1793
1794 self.lbbrackets = ttk.Label(
1795     self.fr_hce,

```

```
1796     text='Brackets spacing').grid(
1797         row=8, column=0, sticky='W', padx=2, pady=5)
1798 self.enbrackets = ttk.Entry(
1799     self.fr_hce,
1800     textvariable=self.varhcebrackets).grid(
1801         row=8, column=1, sticky='W', padx=2, pady=5)
1802
1803 self.lbhceinnerroughness = ttk.Label(
1804     self.fr_hce,
1805     text='HCE inner roughness [ ]').grid(
1806         row=10, column=0, sticky='W', padx=2, pady=5)
1807 self.enhceinnerroughness = ttk.Entry(
1808     self.fr_hce,
1809     textvariable=self.varhceinnerroughness).grid(
1810         row=10, column=1, sticky='W', padx=2, pady=5)
1811
1812 self.lbhceminreynolds = ttk.Label(
1813     self.fr_hce,
1814     text='HCE minimum Reynolds Number [ ]').grid(
1815         row=11, column=0, sticky='W', padx=2, pady=5)
1816 self.enhceminreynolds = ttk.Entry(
1817     self.fr_hce,
1818     textvariable=self.varhceminreynolds).grid(
1819         row=11, column=1, sticky='W', padx=2, pady=5)
1820
1821 self.lbhceabsorptance = ttk.Label(
1822     self.fr_hce,
1823     text='Absorptance [ ]').grid(
1824         row=12, column=0, sticky='W', padx=2, pady=5)
1825 self.enhceabsorptance = ttk.Entry(
1826     self.fr_hce,
1827     textvariable=self.varhceabsorptance).grid(
1828         row=12, column=1, sticky='W', padx=2, pady=5)
1829
1830 self.lbhcetransmittance = ttk.Label(
1831     self.fr_hce,
1832     text='Transmittance [ ]').grid(
1833         row=13, column=0, sticky='W', padx=2, pady=5)
1834 self.enhcetransmittance = ttk.Entry(
1835     self.fr_hce,
1836     textvariable=self.varhcetransmittance).grid(
1837         row=13, column=1, sticky='W', padx=2, pady=5)
1838
1839 self.lbemittanceA0 = ttk.Label(
1840     self.fr_hce,
1841     text='Emittance Factor A0 [ ]').grid(
1842         row=14, column=0, sticky='W', padx=2, pady=5)
1843 self.enemittanceA0 = ttk.Entry(
1844     self.fr_hce,
1845     textvariable=self.varhceemittanceA0).grid(
1846         row=14, column=1, sticky='W', padx=2, pady=5)
1847
1848 self.lbemittanceA1 = ttk.Label(
1849     self.fr_hce,
1850     text='Emittance Factor A1 [ ]').grid(
1851         row=15, column=0, sticky='W', padx=2, pady=5)
1852 self.enemittanceA1 = ttk.Entry(
1853     self.fr_hce,
1854     textvariable=self.varhceemittanceA1).grid(
1855         row=15, column=1, sticky='W', padx=2, pady=5)
```

```
1856
1857
1858 if __name__ == '__main__':
1859
1860     try:
1861         from Tkinter import Tk
1862         import tkMessageBox as messagebox
1863
1864     except ImportError:
1865         from tkinter import Tk
1866         from tkinter import messagebox
1867
1868     interface = Interface()
1869     interface.root.mainloop()
```