```python
# -*- coding: utf-8 -*-

"""
csenergy.py: A Python 3 library for modeling of parabolic-trough solar
collectors
@author: pacomunuera
2020
"""

import numpy as np
import scipy as sc
from CoolProp.CoolProp import PropsSI
import CoolProp.CoolProp as CP
import pandas as pd
import pvlib as pvlib
from tkinter import *
from tkinter.filedialog import askopenfilename
from datetime import datetime, timedelta
import os.path
import matplotlib.pyplot as plt
from matplotlib import rc
import json


class Model:

    def calc_pr(self):
        pass

    def get_hext_eext(self, hce, reext, tro, wind):
        eext = 0.
        hext = 0.

        if hce.parameters['Name'] == 'Solel UVAC 2/2008':
            pass

        elif hce.parameters['Name'] == 'Solel UVAC 3/2010':
            pass

        elif hce.parameters['Name'] == 'Schott PTR70':
            pass

        if (hce.parameters['coating'] == 'CERMET' and
            hce.parameters['annulus'] == 'VACUUM'):

            if wind > 0:
                eext = 1.69E-4*reext**0.0395*tro+1/(11.72+3.45E-6*reext)
                hext = 0.
            else:
                eext = 2.44E-4*tro+0.0832
                hext = 0.
        elif (hce.parameters['coating'] == 'CERMET' and
              hce.parameters['annulus'] == 'NOVACUUM'):
            if wind > 0:
                eext = ((4.88E-10 * reext**0.0395 + 2.13E-4) * tro +
                        1 / (-36 - 1.29E-4 * reext) + 0.0962)
                hext = 2.34 * reext**0.0646
            else:
                eext = 1.97E-4 * tro + 0.0859
                hext = 3.65
```

```
        elif (hce.parameters['coating'] == 'BLACK CHROME' and
              hce.parameters['annulus'] == 'VACUUM'):
            if wind > 0:
                eext = (2.53E-4 * reext**0.0614 * tro +
                        1 / (9.92 + 1.5E-5 * reext))
                hext = 0.
            else:
                eext = 4.66E-4 * tro + 0.0903
                hext = 0.
        elif (hce.parameters['coating'] == 'BLACK CHROME' and
              hce.parameters['annulus'] == 'NOVACUUM'):
            if wind > 0:
                eext = ((4.33E-10 * reext + 3.46E-4) * tro +
                        1 / (-20.5 - 6.32E-4 * reext) + 0.149)
                hext = 2.77 * reext**0.043
            else:
                eext = 3.58E-4 * tro + 0.115
                hext = 3.6

        return hext, eext


class ModelBarbero4thOrder(Model):

    def __init__(self, settings):

        self.parameters = settings
        self.max_err_t = self.parameters['max_err_t']
        self.max_err_tro = self.parameters['max_err_tro']
        self.max_err_pr = self.parameters['max_err_pr']

    def calc_pr(self, hce, htf, row, qabs = None):

        if qabs is None:
            qabs = hce.qabs

        tin = hce.tin

        #  If the hce is the first one in the loop tf = tin, else
        #  tf equals tin plus half the jump of temperature in the previous hce

        if hce.hce_order == 0:
            tf = hce.tin  # HTF bulk temperature
        else:
            tf = hce.tin + 0.5 * (hce.sca.hces[hce.hce_order - 1].tout -
                                  hce.sca.hces[hce.hce_order - 1].tin)

        massflow = hce.sca.loop.massflow
        wspd = row[1]['Wspd']  # Wind speed
        text = row[1]['DryBulb']  # Dry bulb ambient temperature
        sigma = sc.constants.sigma  # Stefan-Bolztmann constant
        dro = hce.parameters['Absorber tube outer diameter']
        dri = hce.parameters['Absorber tube inner diameter']

        L = (hce.parameters['Length'] * hce.parameters['Bellows ratio'] *
             hce.parameters['Shield shading'])

        x = 1  # Calculation grid fits hce longitude

        #  Specific Capacity
```

```
        cp = htf.get_specific_heat(tf, hce.pin)

        #  Internal transmission coefficient.
        # hint = hce.get_hint(tf, hce.pin, htf)

        #  Internal heat transfer coefficient. Eq. 3.22 Barbero2016
        urec = hce.get_urec(tf, hce.pin, htf)

        #  We suppose thermal performance, pr = 1, at first if  the hce is
        #  the first one in the loop or pr_j = pr_j-1 if there is a previous
        #  HCE in the loop.
        pr = hce.get_previous_pr()
        tro1 = tf + qabs * pr / urec

        #  HCE emittance
        eext = hce.get_eext(tro1, wspd)
        #  External Convective Heat Transfer equivalent coefficient
        hext = hce.get_hext(wspd)

        #  Thermal power lost througth  bracktets
        qlost_brackets = hce.get_qlost_brackets(tro1, text)

        #  Thermal power lost. Eq. 3.23 Barbero2016
        # qlost = sigma * eext * (tro1**4 - text**4) + hext * (tro1 - text) + \
        #     qlost_brackets
        qlost = sigma * eext * (tro1**4 - text**4) + hext * (tro1 - text)

        #  Critical Thermal power loss. Eq. 3.50 Barbero2016
        qcrit = sigma * eext * (tf**4 - text**4) + hext * (tf - text)

        #  Critical Internal heat transfer coefficient, Eq. 3.51 Barbero2016
        ucrit = 4 * sigma * eext * tf**3 + hext

        #  Transmission Units Number, Ec. 3.30 Barbero2016
        NTU = urec * x * L * np.pi * dro / (massflow * cp)

        if qabs > 1.1 * qcrit:

            #  We use Barbero2016's simplified model aproximation
            #  Eq. 3.63 Barbero2016
            fcrit = 1 / (1 + (ucrit / urec))

            #  Eq. 3.71 Barbero2016
            pr = fcrit * (1 - qcrit / qabs)

            errtro = 10.
            errpr = 1.
            step = 0

            while ((errtro > self.max_err_tro or errpr > self.max_err_pr) and
                   step < 1000):

                step += 1

                #  Eq. 3.32 Barbero2016
                f0 = qabs / (urec * (tf - text))

                #  Eq. 3.34 Barbero2016
                f1 = ((4 * sigma * eext * text**3) + hext) / urec
                f2 = 6 * (text**2) * (sigma * eext / urec) * (qabs / urec)
```

```
                f3 = 4 * text * (sigma * eext / urec) * ((qabs / urec)**2)
                f4 = (sigma * eext / urec) * ((qabs / urec)**3)

                pr0 = pr

                fx = lambda pr0: (1 - pr0 -
                                  f1 * (pr0 + (1 / f0)) -
                                  f2 * ((pr0 + (1 / f0))**2) -
                                  f3 * ((pr0 + (1 / f0))**3) -
                                  f4 * ((pr0 + (1 / f0))**4))

                dfx = lambda pr0: (-1 - f1 -
                                   2 * f2 * (pr0 + (1 / f0)) -
                                   3 * f3 * (pr0 + (1 / f0))**2 -
                                   4 * f4 * (pr0 + (1 / f0))**3)

                root = sc.optimize.newton(fx,
                                          pr0,
                                          fprime=dfx,
                                          maxiter=100000)

                pr0 = root

                #  Eq. 3.37 Barbero2016
                z = pr0 + (1 / f0)

                #  Eq. 3.40, 3.41 & 3.42 Babero2016
                g1 = 1 + f1 + 2 * f2 * z + 3 * f3 * z**2 + 4 * f4 * z**3
                g2 = 2 * f2 + 6 * f3 * z + 12 * f4 * z**2
                g3 = 6 * f3 + 24 * f4 * z

                #  Eq. 3.39 Barbero2016
                pr2 = ((pr0 * g1 / (1 - g1)) * (1 / (NTU * x)) *
                       (np.exp((1 - g1) * NTU * x / g1) - 1) -
                       (g2 / (6 * g1)) * (pr0 * NTU * x)**2 -
                       (g3 / (24 * g1) * (pr0 * NTU * x)**3))

                errpr = abs(pr2-pr)
                pr = pr2
                hce.pr = pr

                hce.set_tout(htf)
                hce.set_pout(htf)
                tf = 0.5 * (hce.tin + hce.tout)

                #  Specific Capacity
                cp = htf.get_specific_heat(tf, hce.pin)

                #  Internal heat transfer coefficient. Eq. 3.22 Barbero2016
                urec = hce.get_urec(tf, hce.pin, htf)

                #  HCE emittance
                eext = hce.get_eext(tro1, wspd)

                #  External Convective Heat Transfer equivalent coefficient
                hext = hce.get_hext(wspd)

                tro2 = tf + qabs * pr / urec
                errtro = abs(tro2-tro1)
                tro1 = tro2
```

```
241
242                 #  Increase qlost with thermal power lost througth bracktets
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 througth  bracktets
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)
```

```
                errtro = abs(tro2 - tro1)
                tro1 = tro2


class ModelBarbero1stOrder(Model):

    def __init__(self, settings):

        self.parameters = settings
        self.max_err_t = self.parameters['max_err_t']
        self.max_err_tro = self.parameters['max_err_tro']
        self.max_err_pr = self.parameters['max_err_pr']

    def calc_pr(self, hce, htf, row, qabs = None):

        if qabs is None:
            qabs = hce.qabs

        #  If the hce is the first one in the loop tf = tin, else
        #  tf equals tin plus half the jump of temperature in the previous hce
        if hce.hce_order == 0:
            tf = hce.tin  # HTF bulk temperature
        else:
            tf = hce.tin + 0.5 * (hce.sca.hces[hce.hce_order - 1].tout -
                                  hce.sca.hces[hce.hce_order - 1].tin)

        massflow = hce.sca.loop.massflow
        wspd = row[1]['Wspd']  # Wind speed
        text = row[1]['DryBulb']  # Dry bulb ambient temperature
        sigma = sc.constants.sigma  # Stefan-Bolztmann constant
        dro = hce.parameters['Absorber tube outer diameter']
        dri = hce.parameters['Absorber tube inner diameter']
        x = 1  # Calculation grid fits hce longitude

        #  Specific Capacity
        cp = htf.get_specific_heat(tf, hce.pin)

        #  Internal heat transfer coefficient. Eq. 3.22 Barbero2016
        urec = hce.get_urec(tf, hce.pin, htf)
        #  We suppose performance, pr = 1, at first
        pr = 1.0
        tro1 = tf + qabs * pr / urec

        #  HCE emittance
        eext = hce.get_eext(tro1, wspd)
        #  External Convective Heat Transfer equivalent coefficient
        hext = hce.get_hext(wspd)

        #  Thermal power lost througth  bracktets
        qlost_brackets = hce.get_qlost_brackets(tro1, text)

        #  Thermal power lost. Eq. 3.23 Barbero2016
        qlost = sigma * eext * (tro1**4 - text**4) + hext * (tro1 - text)

        #  Critical Thermal power loss. Eq. 3.50 Barbero2016
        qcrit = sigma * eext * (tf**4 - text**4) + hext * (tf - text)

        #  Critical Internal heat transfer coefficient, Eq. 3.51 Barbero2016
        ucrit = 4 * sigma * eext * tf**3 + hext

```

```
        #  Ec. 3.63
        fcrit = 1 / (1 + (ucrit / urec))

        #  Ec. 3.64
        Aext = np.pi * dro * x  # Pendiente de confirmar
        NTUperd = ucrit * Aext / (massflow * cp)

        if qabs > qcrit:

            errtro = 10.
            errpr = 1.
            step = 0

            while ((errtro > self.max_err_tro or errpr > self.max_err_pr) and
                   step < 1000):

                step += 1
                pr2 = ((1 - (qcrit / qabs)) *
                       (1 / (NTUperd * x)) *
                       (1 - np.exp(-NTUperd * fcrit * x)))

                errpr = abs(pr2-pr)
                pr = pr2
                hce.pr = pr

                hce.set_tout(htf)
                hce.set_pout(htf)
                tf = 0.5 * (hce.tin + hce.tout)

               #  Specific Capacity
                cp = htf.get_specific_heat(tf, hce.pin)

                #  Internal heat transfer coefficient. Eq. 3.22 Barbero2016
                urec = hce.get_urec(tf, hce.pin, htf)

                #  HCE emittance
                eext = hce.get_eext(tro1, wspd)

                #  External Convective Heat Transfer equivalent coefficient
                hext = hce.get_hext(wspd)

                tro2 = tf + qabs * pr / urec
                errtro = abs(tro2-tro1)
                tro1 = tro2

                #  Increase qlost with thermal power lost througth bracktets
                qlost_brackets = hce.get_qlost_brackets(tro1, text)

                #  Thermal power loss. Eq. 3.23 Barbero2016
                qlost = sigma * eext * (tro1**4 - text**4)

                #  Critical Thermal power loss. Eq. 3.50 Barbero2016
                qcrit = sigma * eext * (tf**4 - text**4) + hext * (tf - text) + \
                    qlost_brackets

                #  Critical Internal heat transfer coefficient, Eq. 3.51 Barbero2016
                ucrit = 4 * sigma * eext * tf**3 + hext

                #  Ec. 3.63
                fcrit = 1 / (1 + (ucrit / urec))
```

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

```
        self.max_err_pr = self.parameters['max_err_pr']

    def calc_pr(self, hce, htf, row, qabs=None):

        if qabs is None:
            qabs = hce.qabs

        #  If the hce is the first one in the loop tf = tin, else
        #  tf equals tin plus half the jump of temperature in the previous hce
        if hce.hce_order == 0:
            tf = hce.tin  # HTF bulk temperature
        else:
            tf = hce.tin + 0.5 * (hce.sca.hces[hce.hce_order - 1].tout -
                                  hce.sca.hces[hce.hce_order - 1].tin)

        wspd = row[1]['Wspd']  # Wind speed
        text = row[1]['DryBulb']  # Dry bulb ambient temperature
        sigma = sc.constants.sigma  # Stefan-Bolztmann constant
        dro = hce.parameters['Absorber tube outer diameter']
        dri = hce.parameters['Absorber tube inner diameter']
        x = 1  # Calculation grid fits hce longitude

        #  Specific Capacity
        cp = htf.get_specific_heat(tf, hce.pin)

        #  Internal heat transfer coefficient. Eq. 3.22 Barbero2016
        urec = hce.get_urec(tf, hce.pin, htf)

        #  We suppose performance, pr = 1, at first
        pr = 1.0
        tro1 = tf + qabs * pr / urec

        #  HCE emittance
        eext = hce.get_eext(tro1, wspd)
        #  External Convective Heat Transfer equivalent coefficient
        hext = hce.get_hext(wspd)

        #  Thermal power lost througth  bracktets
        qlost_brackets = hce.get_qlost_brackets(tro1, text)

        #  Thermal power loss. Eq. 3.23 Barbero2016
        qlost = sigma * eext * (tro1**4 - text**4) + hext * (tro1 - text)

        #  Critical Thermal power loss. Eq. 3.50 Barbero2016
        qcrit = sigma * eext * (tf**4 - text**4) + hext * (tf - text)

        #  Critical Internal heat transfer coefficient, Eq. 3.51 Barbero2016
        ucrit = 4 * sigma * eext * tf**3 + hext

        #  Ec. 3.63
        ## fcrit = (1 / ((4 * eext * tfe**3 / urec) + (hext / urec) + 1))
        fcrit = 1 / (1 + (ucrit / urec))

        if qabs > qcrit:

            errtro = 10.
            errpr = 1.
            step = 0

            while ((errtro > self.max_err_tro or errpr > self.max_err_pr) and
```

```
                   step < 1000):

                step += 1
                pr2 = fcrit * (1 - (qcrit / qabs))


                errpr = abs(pr2-pr)
                pr = pr2
                hce.pr = pr

                hce.set_tout(htf)
                hce.set_pout(htf)
                tf = 0.5 * (hce.tin + hce.tout)

                #  Specific Capacity
                cp = htf.get_specific_heat(tf, hce.pin)

                #  Internal heat transfer coefficient. Eq. 3.22 Barbero2016
                urec = hce.get_urec(tf, hce.pin, htf)

                #  We suppose performance, pr = 1, at first
                tro1 = tf + qabs * pr / urec

                #  HCE emittance
                eext = hce.get_eext(tro1, wspd)

                #  External Convective Heat Transfer equivalent coefficient
                hext = hce.get_hext(wspd)

                tro2 = tf + qabs * pr / urec
                errtro = abs(tro2-tro1)
                tro1 = tro2

                #  Thermal power lost througth  bracktets
                qlost_brackets = hce.get_qlost_brackets(tro1, text)

                #  Thermal power loss. Eq. 3.23 Barbero2016
                qlost = sigma * eext * (tro1**4 - text**4) + hext * (tro1 - text) +
\
                    qlost_brackets

                #  Critical Thermal power loss. Eq. 3.50 Barbero2016
                qcrit = sigma * eext * (tf**4 - text**4) + hext * (tf - text)

                #  Critical Internal heat transfer coefficient, Eq. 3.51 Barbero2016
                ucrit = 4 * sigma * eext * tf**3 + hext

                #  Ec. 3.63
                ## fcrit = (1 / ((4 * eext * tfe**3 / urec) + (hext / urec) + 1))
                fcrit = 1 / (1 + (ucrit / urec))

            if step == 1000:
                print('No se alcanzó convergencia. HCE', hce.get_index())
                print(qabs, qcrit, urec, ucrit)

            hce.pr = hce.pr * (1 - qlost_brackets / qabs)
            hce.qlost = qlost
            hce.qlost_brackets = qlost_brackets

        else:
```

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

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

```
            fx = lambda x: x + 2 * np.log10(a + b * x )

            dfx = lambda x: 1 + (2 * b) / (np.log(10) * (a + b * x))

            root = sc.optimize.newton(fx,
                                      x,
                                      fprime=dfx,
                                      maxiter=10000)

            darcy_factor = 1 / (root**2)

        rho = htf.get_density(self.tin, self.pin)
        v = 4 * self.sca.loop.massflow / (rho * np.pi * D**2)
        g = sc.constants.g

        # Ec. Darcy-Weisbach para el cálculo de la pérdida de carga
        deltap_mcl = darcy_factor * (self.parameters['Length'] / D) * \
            (v**2 / (2 * g))
        deltap = deltap_mcl * rho * g
        self.pout = self.pin - deltap
        '''

        self.pout = self.pin

    def set_pr_opt(self, solarpos):

        IAM = self.sca.get_IAM(solarpos)
        aoi = self.sca.get_aoi(solarpos)
        pr_opt_peak = self.get_pr_opt_peak()
        self.pr_opt = pr_opt_peak * IAM * np.cos(np.radians(aoi))

    def set_qabs(self, aoi, solarpos, row):
        """Total solar power that reach de absorber tube per longitude unit."""
        dni = row[1]['DNI']
        cg = (self.sca.parameters['Aperture'] /
              (np.pi*self.parameters['Absorber tube outer diameter']))

        pr_shadows = self.get_pr_shadows2(solarpos)
        pr_borders = self.get_pr_borders(aoi)
        #  Ec. 3.20 Barbero
        self.qabs = self.pr_opt * cg * dni * pr_borders * pr_shadows

    def get_krec(self, t):

        # From Choom S. Kim for A316
        #  kt = 100*(0.1241+0.00003279*t)

        # Ec. 4.22 Conductividad para el acero 321H.
        kt = 0.0153 * (t - 273.15) + 14.77

        return kt

    def get_urec(self, t, p, htf):

        #  HCE wall thermal conductivity
        krec = self.get_krec(t)

        #  Specific Capacity
        cp = htf.get_specific_heat(t, p)

```

```
        #  Internal transmission coefficient.
        hint = self.get_hint(t, p, htf)

        #  Internal heat transfer coefficient. Eq. 3.22 Barbero2016
        return (1 / ((1 / hint) + (
            self.parameters['Absorber tube outer diameter'] *
            np.log(self.parameters['Absorber tube outer diameter'] /
                   self.parameters['Absorber tube inner diameter'])) /
            (2 * krec)))

    def get_pr_opt_peak(self):

        alpha = self.get_absorptance()
        tau = self.get_transmittance()
        rho = self.sca.parameters['Reflectance']
        gamma = self.sca.get_solar_fraction()

        pr_opt_peak = alpha * tau * rho * gamma

        if pr_opt_peak > 1 or pr_opt_peak < 0:
            print("ERROR", pr_opt_peak)

        return pr_opt_peak

    def get_pr_borders(self, aoi):

        if aoi > 90:
            pr_borders = 0.0

        else:
            sca_unused_length = (self.sca.parameters["Focal Len"] *
                                 np.tan(np.radians(aoi)))

            unused_hces = sca_unused_length // self.parameters["Length"]

            unused_part_of_hce = ((sca_unused_length %
                                   self.parameters["Length"]) /
                                  self.parameters["Length"])

            if self.hce_order < unused_hces:
                pr_borders = 0.0

            elif self.hce_order == unused_hces:
                pr_borders = 1 - \
                    ((sca_unused_length % self.parameters["Length"]) /
                     self.parameters["Length"])
            else:
                pr_borders = 1.0

            if pr_borders > 1.0 or pr_borders < 0.0:
                print("ERROR pr_bordes out of limits", pr_borders)

        return pr_borders

    def get_pr_shadows(self, solarpos):

        if solarpos['elevation'][0] < 0:
            shading = 1

        else:
```

```
            shading = 1 - (np.sin(np.radians(abs(solarpos['elevation'][0]))) *
                      self.sca.loop.parameters['row_spacing'] /
                      self.sca.parameters['Aperture'])

        if shading < 0.0 or shading > 1.0:
            shading = 0.0

        shading = 1 - shading

        if  shading > 1 or  shading < 0:
            print("ERROR shading",  shading)

        s2 = self.get_pr_shadows2(solarpos)

        return shading

    def get_pr_shadows2(self, solarpos):

        beta0 = self.sca.get_tracking_angle(solarpos)

        if beta0 >= 0:
            sigmabeta = 0
        else:
            sigmabeta = 1

        #  Surface tilt
        beta = beta0 + 180 * sigmabeta

        surface_azimuth = self.sca.get_surface_azimuth(solarpos)

        Ls = abs(len(self.sca.loop.scas) * self.sca.parameters['SCA Length'] -
                 abs(self.sca.loop.parameters['row_spacing'] *
                     np.tan(np.radians(surface_azimuth -
                                       solarpos['azimuth'][0]))))

        ls = Ls / (len(self.sca.loop.scas) * self.sca.parameters['SCA Length'])

        if solarpos['zenith'][0] < 90:
            shading = min(abs(np.cos(np.radians(beta0))) * \
                  self.sca.loop.parameters['row_spacing'] / \
                      self.sca.parameters['Aperture'], 1)
        else:
            shading = 0

        return shading

    def get_hext(self, wspd):

        #  TO-DO:
        return 0.0

    def get_hint(self, t, p, fluid):

        #  Gnielinski correlation. Eq. 4.15 Barbero2016
        kf = fluid.get_thermal_conductivity(t, p)
        dri = self.parameters['Absorber tube inner diameter']

        #  Prandtl number
        prf = fluid.get_prandtl(t, p)

```

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

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

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

```
                surface_azimuth = 90  # Surface facing east
            else:
                surface_azimuth = 270  # Surface facing west
        elif self.loop.parameters['Tracking Type'] == 2:  # E-W single axis
            surface_azimuth = 180  # Surface facing the equator

        return surface_azimuth


class __Loop__(object):


    def __init__(self, settings):

        self.scas = []
        self.parameters = settings

        self.tin = 0.0
        self.tout = 0.0
        self.pin = 0.0
        self.pout = 0.0
        self.massflow = 0.0
        self.qabs = 0.0
        self.qlost = 0.0
        self.qlost_brackets = 0.0
        self.wasted_power = 0.0
        self.pr = 0.0
        self.pr_opt = 0.0

        self.act_tin = 0.0
        self.act_tout = 0.0
        self.act_pin = 0.0
        self.act_pout = 0.0
        self.act_massflow = 0.0  # Actual massflow measured by the flowmeter

        self.wasted_power = 0.0
        self.tracking = True

    def initialize(self, type_of_source, source=None):

        if type_of_source == 'rated':
            self.massflow = self.parameters['rated_massflow']
            self.tin = self.parameters['rated_tin']
            self.pin = self.parameters['rated_pin']

        elif type_of_source == 'subfield' and source is not None:
            self.massflow = source.massflow / len(source.loops)
            self.tin = source.tin
            self.pin = source.pin

        elif type_of_source  == 'solarfield' and source is not None:
            self.massflow = source.massflow / source.total_loops
            self.tin = solarfield.tin
            self.pin = solarfield.pin

        elif type_of_source == 'values' and source is not None:
            self.massflow = source['massflow']
            self.tin = source['tin']
            self.pin = source['pin']

```

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

```
        self.pr_opt = base_loop.pr_opt
        self.qabs = base_loop.qabs
        self.qlost = base_loop.qlost
        self.qlost_brackets = base_loop.qlost_brackets

    def calc_loop_pr_for_massflow(self, row, solarpos, htf, model):

        for s in self.scas:
            aoi = s.get_aoi(solarpos)
            for h in s.hces:
                h.set_pr_opt(solarpos)
                h.set_qabs(aoi, solarpos, row)
                h.set_tin()
                h.set_pin()
                model.calc_pr(h, htf, row)

        self.tout = self.scas[-1].hces[-1].tout
        self.pout = self.scas[-1].hces[-1].pout
        self.set_loop_values_from_HCEs()
        self.set_wasted_power(htf)

    def calc_loop_pr_for_tout(self, row, solarpos, htf, model):

        dri = self.scas[0].hces[0].parameters['Absorber tube inner diameter']
        min_reynolds = self.scas[0].hces[0].parameters['Min Reynolds']

        min_massflow = htf.get_massflow_from_Reynolds(dri, self.tin, self.pin,
                                                      min_reynolds)

        max_error = model.max_err_t  # % desviation tolerance
        search = True
        step = 0

        while search:

            self.calc_loop_pr_for_massflow(row, solarpos, htf, model)

            err = abs(self.tout-self.parameters['rated_tout'])

            if err > max_error and step <1000:
                step += 1
                if self.tout >= self.parameters['rated_tout']:
                    self.massflow *= (1 + err / self.parameters['rated_tout'])
                    search = True
                elif (self.massflow > min_massflow and
                      self.massflow >
                      self.parameters['min_massflow']):
                    self.massflow *= (1 - err / self.parameters['rated_tout'])
                    search = True
                else:
                    self.massflow = max(
                        min_massflow,
                        self.parameters['min_massflow'])
                    self.calc_loop_pr_for_massflow(row, solarpos, htf, model)
                    search = False
            else:
                search = False
                if step>=1000:
                    print("Massflow convergence failed")

```

```
        self.tout = self.scas[-1].hces[-1].tout
        self.pout = self.scas[-1].hces[-1].pout
        self.set_loop_values_from_HCEs()
        self.wasted_power = 0.0


    def check_min_massflow(self, htf):

        dri = self.parameters['Absorber tube inner diameter']
        t = self.tin
        p = self.pin
        re = self.parameters['Min Reynolds']
        loop_min_massflow = htf.get_massflow_from_Reynolds(
                dri, t, p , re)

        if  self.massflow < loop_min_massflow:
            print("Too low massflow", self.massflow ,"<",
                  loop_min_massflow)

    def set_wasted_power(self, htf):

        if self.tout > self.parameters['tmax']:
            self.wasted_power = self.massflow * htf.get_delta_enthalpy(
                self.parameters['tmax'], self.tout, self.pin, self.pout)
        else:
            self.wasted_power = 0.0

    def get_values(self, type = None):

        _values = {}

        if type is None:
            _values =  {'tin': self.tin, 'tout': self.tout,
                        'pin': self.pin, 'pout': self.pout,
                        'mf': self.massflow}
        elif type =='required':
            _values =  {'tin': self.tin, 'tout': self.tout,
                        'pin': self.pin, 'pout': self.pout,
                        'req_mf': self.req_massflow}
        elif type =='actual':
            _values =  {'actual_tin': self.actual_tin, 'tout': self.tout,
                        'pin': self.pin, 'pout': self.pout,
                        'mf': self.req_massflow}

        return _values

    def get_id(self):

        return 'LP.{0}.{1:03d}'.format(self.subfield.name, self.loop_order)

class Loop(__Loop__):

    def __init__(self, subfield, loop_order, settings):

        self.subfield = subfield
        self.loop_order = loop_order

        super().__init__(settings)
```

```
class BaseLoop(__Loop__):

    def __init__(self, settings, sca_settings, hce_settings):

        super().__init__(settings)

        self.parameters_sca = sca_settings
        self.parameters_hce = hce_settings

        for s in range(settings['scas']):
            self.scas.append(SCA(self, s, sca_settings))
            for h in range(settings['hces']):
                self.scas[-1].hces.append(
                    HCE(self.scas[-1], h, hce_settings))


    def get_id(self, subfield = None):

        id = ''
        if subfield is not None:
            id = 'LB.'+subfield.name
        else:
            id = 'LB.000'
        return id

    def get_qlost_brackets(self, tf, text):

        #  Ec. 4.12

        L = self.parameters_hce['Length']
        n = (L / self.parameters_hce['Brackets'])
        pb = 0.2032
        acsb = 1.613e-4
        kb = 48
        hb = 20
        tbase = tf - 10

        return n * (np.sqrt(pb * kb * acsb * hb) * (tbase - text)) / L

    def get_pr_opt_peak(self):

        alpha = self.parameters_hce['Absorber absorptance']
        tau = self.parameters_hce['Envelope transmittance']
        rho = self.parameters_sca['Reflectance']
        gamma = self.get_solar_fraction()

        pr_opt_peak = alpha * tau * rho * gamma

        if pr_opt_peak > 1 or pr_opt_peak < 0:
            print("ERROR pr_opt_peak", pr_opt_peak)

        return pr_opt_peak


    def get_pr_borders(self, aoi):

        if aoi > 90:
            pr_borders = 0.0

        else:
```

```
            sca_unused_length = (self.parameters_sca["Focal Len"] *
                                 np.tan(np.radians(aoi)))

            pr_borders = 1 - sca_unused_length / \
                (self.parameters['hces'] * self.parameters_hce["Length"])

            if pr_borders > 1.0 or pr_borders < 0.0:
                print("ERROR pr_geo out of limits", pr_borders)

        return pr_borders


    def get_pr_shadows(self, solarpos):

        if solarpos['elevation'][0] < 0:
            shading = 1

        else:
            shading = 1 - (np.sin(np.radians(abs(solarpos['elevation'][0]))) *
                     self.parameters['row_spacing'] /
                     self.parameters_sca['Aperture'])

        if shading < 0.0 or shading > 1.0:
            shading = 0.0


        shading = 1 - shading


        if  shading > 1 or  shading < 0:
            print("ERROR shading",  shading)

        return shading

    def get_pr_shadows2(self, solarpos):

        beta0 = self.get_tracking_angle(solarpos)

        if beta0 >= 0:
            sigmabeta = 0
        else:
            sigmabeta = 1

        #  Surface tilt
        beta = beta0 + 180 * sigmabeta

        surface_azimuth = self.get_surface_azimuth(solarpos)

        Ls = abs(len(self.scas) * self.parameters_sca['SCA Length'] -
                 abs(self.parameters['row_spacing'] *
                     np.tan(np.radians(surface_azimuth -
                                       solarpos['azimuth'][0]))))

        ls = Ls / (len(self.scas) * self.parameters_sca['SCA Length'])

        if solarpos['zenith'][0] < 90:
            shading = min(abs(np.cos(np.radians(beta0))) * \
                  self.parameters['row_spacing'] / \
                      self.parameters_sca['Aperture'], 1)
        else:
```

```
            shading = 0

        return shading


    def get_solar_fraction(self):

        #  Cleanliness two times because it affects mirror and envelope
        solarfraction = (self.parameters_sca['Geom.Accuracy'] *
                         self.parameters_sca['Track Twist'] *
                         self.parameters_sca['Cleanliness'] *
                         self.parameters_sca['Cleanliness'] *
                         self.parameters_sca['Factor'] *
                         self.parameters_sca['Availability'])

        if solarfraction > 1 or solarfraction < 0:
            print("ERROR",  solarfraction)

        return solarfraction

    def get_IAM(self, solarpos):

        if  solarpos['zenith'][0] > 80:
            kiam = 0.0
        else:

            aoi = self.get_aoi(solarpos)

            F0 = self.parameters_sca['IAM Coefficient F0']
            F1 = self.parameters_sca['IAM Coefficient F1']
            F2 = self.parameters_sca['IAM Coefficient F2']

            if (aoi > 0 and aoi < 80):
                kiam = (F0 + (F1 * np.radians(aoi) + F2 * np.radians(aoi)**2) /
                        np.cos(np.radians(aoi)))

            if kiam > 1.0:
                kiam = 1.0

        if  kiam > 1.0 or  kiam < 0.0:
            print("ERROR",  kiam, aoi)

        return kiam


    def get_aoi(self, solarpos):

        sigmabeta = 0.0
        beta0 = 0.0

        surface_azimuth = self.get_surface_azimuth(solarpos)
        beta0 = self.get_tracking_angle(solarpos)

        if beta0 >= 0:
            sigmabeta = 0
        else:
            sigmabeta = 1

        beta = beta0 + 180 * sigmabeta
        aoi = pvlib.irradiance.aoi(beta,
```

```
                                    surface_azimuth,
                                    solarpos['zenith'][0],
                                    solarpos['azimuth'][0])
        return aoi


    def get_tracking_angle(self, solarpos):

        surface_azimuth = self.get_surface_azimuth(solarpos)
        #  Tracking angle for a collector with tilt = 0
        #  Ec. 2.32 Technical Manual for the SAM Physical Trough Model
        beta0 = np.degrees(
                    np.arctan(np.tan(np.radians(solarpos['zenith'][0])) *
                              np.cos(np.radians(surface_azimuth -
                                                solarpos['azimuth'][0]))))
        return beta0


    def get_surface_azimuth(self, solarpos):

        if self.parameters['Tracking Type'] == 1: # N-S single axis tracker
            if solarpos['azimuth'][0] > 0 and solarpos['azimuth'][0] <= 180:
                surface_azimuth = 90 # Surface facing east
            else:
                surface_azimuth = 270 # Surface facing west
        elif self.parameters['Tracking Type'] == 2:  # E-W single axis tracker
            surface_azimuth = 180  # Surface facing the equator

        return surface_azimuth


class Subfield(object):
    '''
    Parabolic Trough Solar Field
    '''

    def __init__(self, solarfield, settings):

        self.solarfield = solarfield
        self.name = settings['name']
        self.parameters = settings
        self.loops = []

        self.tin = 0.0
        self.tout = 0.0
        self.pin = 0.0
        self.pout = 0.0
        self.massflow = 0.0
        self.qabs = 0.0
        self.qlost = 0.0
        self.qlost_brackets = 0.0
        self.wasted_power = 0.0
        self.pr = 0.0
        self.pr_opt = 0.0

        self.act_tin = 0.0
        self.act_tout = 0.0
        self.act_pin = 0.0
        self.act_pout = 0.0
        self.act_massflow = 0.0
```

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

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

```
        self.qabs = 0.0
        self.qlost = 0.0
        self.qlost_brackets = 0.0
        self.wasted_power = 0.0
        self.pr = 0.0
        self.pr_opt = 0.0
        self.pwr = 0.0

        self.act_tin = 0.0
        self.act_tout = 0.0
        self.act_pin = 0.0
        self.act_pout = 0.0
        self.act_massflow = 0.0
        self.act_pwr = 0.0

        self.rated_tin = loop_settings['rated_tin']
        self.rated_tout = loop_settings['rated_tout']
        self.rated_pin = loop_settings['rated_pin']
        self.rated_pout = loop_settings['rated_pout']
        self.rated_massflow = (loop_settings['rated_massflow'] *
                               self.total_loops)

        for sf in subfield_settings:
            self.total_loops += sf['loops']
            self.subfields.append(Subfield(self, sf))
            for l in range(sf['loops']):
                self.subfields[-1].loops.append(
                    Loop(self.subfields[-1], l, loop_settings))
                for s in range(loop_settings['scas']):
                    self.subfields[-1].loops[-1].scas.append(
                        SCA(self.subfields[-1].loops[-1], s, sca_settings))
                    for h in range (loop_settings['hces']):
                        self.subfields[-1].loops[-1].scas[-1].hces.append(
                            HCE(self.subfields[-1].loops[-1].scas[-1], h,
                                hce_settings))

        # TO-DO: FUTURE WORK
        self.storage_available = False
        self.operation_mode = "subfield_heating"

    def initialize(self, source, values = None):

        if source == 'rated':
            self.massflow = self.rated_massflow
            self.tin = self.rated_tin
            self.pin = self.rated_pin
            self.tout = self.rated_tout
            self.pout = self.rated_pout

        elif source == 'actual':
            self.massflow = self.act_massflow
            self.tin = self.act_tin
            self.pin = self.act_pin
            self.tout = self.act_tout
            self.pout = self.act_pout

        elif source == 'values' and values is not None:
            self.massflow = values['massflow']
            self.tin = values['tin']
            self.pin = values['pin']
```

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

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

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

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

```
                    self.simulate_solarfield(solarpos, row)
                    self.solarfield.set_thermal_power(self.htf, self.datatype)
                    self.gather_simulation_data(row)

                    str_data = ("SIMULATION: {0} " +
                                "DNI: {1:3.0f} W/m2 Qm: {2:4.1f}kg/s " +
                                "Tin: {3:4.1f}° Tout: {4:4.1f}°C")

                    print(str_data.format(row[0],
                                          row[1]['DNI'],
                                          self.solarfield.massflow,
                                          self.solarfield.tin - 273.15,
                                          self.solarfield.tout - 273.15))


                if self.benchmark and self.datatype == 2:  # 2: Field Data File available
                    self.benchmark_solarfield(solarpos, row)
                    self.solarfield.set_thermal_power(self.htf, self.datatype)
                    self.gather_benchmark_data(row)

                    str_data = ("BENCHMARK: {0} " +
                                "DNI: {1:3.0f} W/m2 act_Qm: {2:4.1f}kg/s " +
                                "act_Tin: {3:4.1f}° act_Tout: {4:4.1f}° " +
                                "Tout: {5:4.1f}°")

                    print(str_data.format(row[0],
                                          row[1]['DNI'],
                                          self.solarfield.act_massflow,
                                          self.solarfield.act_tin - 273.15,
                                          self.solarfield.act_tout - 273.15,
                                          self.solarfield.tout - 273.15))

        self.save_results()

    def simulate_solarfield(self, solarpos, row):

        if self.datatype == 1:
            for s in self.solarfield.subfields:
                s.initialize('rated')
                for l in s.loops:
                    l.initialize('rated')
            self.solarfield.initialize('rated')
            self.base_loop.initialize('rated')
            if self.fastmode:
                if solarpos['zenith'][0] > 90:
                    self.base_loop.massflow = \
                        self.base_loop.parameters['min_massflow']
                    self.base_loop.calc_loop_pr_for_massflow(
                        row, solarpos, self.htf, self.model)
                else:
                    self.base_loop.calc_loop_pr_for_tout(
                        row, solarpos, self.htf, self.model)
                for s in self.solarfield.subfields:
                    for l in s.loops:
                        l.load_from_base_loop(self.base_loop)
            else:
                for s in self.solarfield.subfields:
                    for l in s.loops:
                        if solarpos['zenith'][0] > 90:
```

```
                            l.massflow = \
                                self.base_loop.parameters['min_massflow']
                            l.calc_loop_pr_for_massflow(
                                row, solarpos, self.htf, self.model)
                        else:
                            if l.loop_order > 1:
                                # For a faster convergence
                                l.massflow = \
                                l.subfield.loops[l.loop_order - 1].massflow
                            l.calc_loop_pr_for_tout(
                                row, solarpos, self.htf, self.model)

        elif self.datatype == 2:
            #  1st, we initialize subfields because actual data are given for
            #  subfields. 2nd, we initialize solarfield.
            for s in self.solarfield.subfields:
                s.load_actual(row)
                s.initialize('actual')
            self.solarfield.load_actual(self.htf)
            self.solarfield.initialize('actual')
            if self.fastmode:
                # Force minimum massflow at night
                for s in self.solarfield.subfields:
                    self.base_loop.initialize('subfield', s)
                    if solarpos['zenith'][0] > 90:
                        self.base_loop.massflow = \
                            self.base_loop.parameters['min_massflow']
                        self.base_loop.calc_loop_pr_for_massflow(
                            row, solarpos, self.htf, self.model)
                    else:
                        self.base_loop.calc_loop_pr_for_tout(
                            row, solarpos, self.htf, self.model)
                    for l in s.loops:
                        l.load_from_base_loop(self.base_loop)
            else:
                for s in self.solarfield.subfields:
                    for l in s.loops:
                        # l.load_actual(s)
                        l.initialize('subfield', s)
                        if solarpos['zenith'][0] > 90:
                            l.massflow = l.parameters['min_massflow']
                            l.calc_loop_pr_for_massflow(
                                row, solarpos, self.htf, self.model)
                        else:
                            if l.loop_order > 1:
                                # For a faster convergence
                                l.massflow = \
                                l.subfield.loops[l.loop_order - 1].massflow
                            l.calc_loop_pr_for_tout(
                                row, solarpos, self.htf, self.model)

        for s in self.solarfield.subfields:
            s.set_subfield_values_from_loops(self.htf)

        self.solarfield.set_solarfield_values_from_subfields(self.htf)

    def benchmark_solarfield(self, solarpos, row):

        for s in self.solarfield.subfields:
            s.load_actual(row)
```

```
            s.initialize('actual')
        self.solarfield.load_actual(self.htf)
        self.solarfield.initialize('actual')

        if self.fastmode:
            for s in self.solarfield.subfields:
                self.base_loop.initialize('subfield', s)
                self.base_loop.calc_loop_pr_for_massflow(
                    row, solarpos, self.htf, self.model)
                self.base_loop.set_loop_values_from_HCEs()

                for l in s.loops:
                    l.load_from_base_loop(self.base_loop)

                s.set_subfield_values_from_loops(self.htf)

        else:
            for s in self.solarfield.subfields:
                for l in s.loops:
                    # l.load_actual()
                    l.initialize('subfield', s)
                    l.calc_loop_pr_for_massflow(
                        row, solarpos, self.htf, self.model)
                    l.set_loop_values_from_HCEs()

                s.set_subfield_values_from_loops(self.htf)

        self.solarfield.set_solarfield_values_from_subfields(self.htf)


    def store_values(self, row, values):

        for v in values:
            self.datasource.dataframe.at[row[0], v] = values[v]

    def gather_general_data(self, row, solarpos):

        self.datasource.dataframe.at[row[0], 'elevation'] = \
            solarpos['elevation'][0]
        self.datasource.dataframe.at[row[0], 'zenith'] = \
            solarpos['zenith'][0]
        self.datasource.dataframe.at[row[0], 'azimuth'] = \
            solarpos['azimuth'][0]
        aoi = self.base_loop.get_aoi(solarpos)
        self.datasource.dataframe.at[row[0], 'aoi'] = aoi
        self.datasource.dataframe.at[row[0], 'IAM'] = \
            self.base_loop.get_IAM(solarpos)
        self.datasource.dataframe.at[row[0], 'pr_shadows'] = \
            self.base_loop.get_pr_shadows2(solarpos)
        self.datasource.dataframe.at[row[0], 'pr_borders'] = \
            self.base_loop.get_pr_borders(aoi)
        self.datasource.dataframe.at[row[0], 'pr_opt_peak'] = \
            self.base_loop.get_pr_opt_peak()
        self.datasource.dataframe.at[row[0], 'solar_fraction'] = \
            self.base_loop.get_solar_fraction()

    def gather_simulation_data(self, row):

        #  Solarfield data
        self.datasource.dataframe.at[row[0], 'SF.x.mf'] = \
```

```
            self.solarfield.massflow
        self.datasource.dataframe.at[row[0], 'SF.x.tin'] = \
            self.solarfield.tin - 273.15
        self.datasource.dataframe.at[row[0], 'SF.x.tout'] = \
            self.solarfield.tout - 273.15
        self.datasource.dataframe.at[row[0], 'SF.x.pin'] = \
            self.solarfield.pin
        self.datasource.dataframe.at[row[0], 'SF.x.pout'] = \
            self.solarfield.pout
        self.datasource.dataframe.at[row[0], 'SF.x.prth'] = \
            self.solarfield.pr
        self.datasource.dataframe.at[row[0], 'SF.x.prop'] = \
            self.solarfield.pr_opt
        self.datasource.dataframe.at[row[0], 'SF.x.qabs'] = \
            self.solarfield.qabs
        self.datasource.dataframe.at[row[0], 'SF.x.qlst'] = \
            self.solarfield.qlost
        self.datasource.dataframe.at[row[0], 'SF.x.qlbk'] = \
            self.solarfield.qlost_brackets
        self.datasource.dataframe.at[row[0], 'SF.x.pwr'] = \
            self.solarfield.pwr

        if  self.datatype == 2:
            if row[1]['GrossPower']>0:
                self.datasource.dataframe.at[row[0], 'SF.x.globalpr'] = \
                    row[1]['GrossPower'] / self.solarfield.pwr
            else:
                self.datasource.dataframe.at[row[0], 'SF.x.globalpr'] = 0
        else:
            self.datasource.dataframe.at[row[0], 'SF.x.globalpr'] = 0

        if self.fastmode:

            values = {
                self.base_loop.get_id() + '.x.mf': self.base_loop.massflow,
                self.base_loop.get_id() + '.x.tin':
                    self.base_loop.tin - 273.15,
                self.base_loop.get_id() + '.x.tout':
                    self.base_loop.tout  - 273.15,
                self.base_loop.get_id() + '.x.pin': self.base_loop.pin,
                self.base_loop.get_id() + '.x.pout': self.base_loop.pout,
                self.base_loop.get_id() + '.x.prth': self.base_loop.pr,
                self.base_loop.get_id() + '.x.prop': self.base_loop.pr_opt,
                self.base_loop.get_id() + '.x.qabs': self.base_loop.qabs,
                self.base_loop.get_id() + '.x.qlst': self.base_loop.qlost,
                self.base_loop.get_id() + '.x.qlbk': \
                    self.base_loop.qlost_brackets}
            self.store_values(row, values)

        for s in self.solarfield.subfields:
            #  Agretate data from subfields
            values = {
                s.get_id() + '.x.mf': s.massflow,
                s.get_id() + '.x.tin': s.tin  - 273.15,
                s.get_id() + '.x.tout': s.tout - 273.15,
                s.get_id() + '.x.pin': s.pin,
                s.get_id() + '.x.pout': s.pout,
                s.get_id() + '.x.prth': s.pr,
                s.get_id() + '.x.prop': s.pr_opt,
                s.get_id() + '.x.qabs': s.qabs,
```

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

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

```
                    values = {
                        l.get_id() + '.a.mf':  l.act_massflow,
                        l.get_id() + '.a.tin':  l.act_tin - 273.15,
                        l.get_id() + '.a.tout':  l.act_tout - 273.15,
                        l.get_id() + '.b.tout':  l.tout - 273.15,
                        l.get_id() + '.a.pin':  l.act_pin,
                        l.get_id() + '.a.pout':  l.act_pout,
                        l.get_id() + '.b.pout':  l.pout,
                        l.get_id() + '.b.prth':  l.pr,
                        l.get_id() + '.b.prop':  l.pr_opt,
                        l.get_id() + '.b.qabs':  l.qabs,
                        l.get_id() + '.b.qlost':  l.qlost,
                        l.get_id() + '.b.qlbk':  l.qlost_brackets,
                        l.get_id() + '.b.wpwr': l.wasted_power}

                    self.store_values(row, values)


    def show_report(self, keys= None):

        self.report_df[keys].plot(
                        figsize=(20,10), linewidth=5, fontsize=20)
        plt.xlabel('Date', fontsize=20)
        pd.set_option('display.max_rows', None)
        pd.set_option('display.max_columns', None)
        pd.set_option('display.width', None)

    def save_results(self):

        keys = ['DNI','elevation','zenith','azimuth','aoi','IAM',
                'pr_shadows','pr_borders','pr_opt_peak','solar_fraction']

        keys_graphics_power = ['DNI']
        keys_graphics_temp = ['DNI']

        keys_a = ['NetPower','AuxPower','GrossPower',
                  'SF.a.mf','SF.a.tin', 'SF.a.tout',
                  'SF.a.pwr','SF.a.prth','SF.a.globalpr']
        keys_graphics_power_a = ['NetPower','AuxPower','GrossPower',
                                 'SF.a.pwr']
        keys_graphics_temp_a = ['SF.a.mf','SF.a.tin', 'SF.a.tout']

        keys_x = ['SF.x.mf','SF.x.tin','SF.x.tout', 'SF.x.pwr',
                  'SF.x.prth','SF.x.globalpr']
        keys_graphics_power_x = ['SF.x.pwr']
        keys_graphics_temp_x = ['SF.x.mf', 'SF.x.tout']

        keys_b = ['SF.b.tout','SF.b.pwr', 'SF.b.wpwr',
                  'SF.b.prth','SF.b.globalpr']
        keys_graphics_power_b = ['SF.b.pwr']
        keys_graphics_temp_b = ['SF.b.tout']

        if self.datatype == 2:
            keys += keys_a
            keys_graphics_power += keys_graphics_power_a
            keys_graphics_temp += keys_graphics_temp_a

        if self.simulation == True:
            keys += keys_x
            keys_graphics_power += keys_graphics_power_x
```

```
            keys_graphics_temp += keys_graphics_temp_x

        if self.benchmark == True:
            keys += keys_b
            keys_graphics_power += keys_graphics_power_b
            keys_graphics_temp += keys_graphics_temp_b

        self.report_df = self.datasource.dataframe

        try:
            initialdir = "./simulations_outputs/"
            prefix = datetime.today().strftime("%Y%m%d %H%M%S ")
            filename_complete = str(self.ID) + "_COMPLETE"
            filename_report = str(self.ID) + "_REPORT"
            sufix = ".csv"

            path_complete = initialdir + prefix + filename_complete + sufix
            path_report = initialdir + prefix + filename_report + sufix

            self.datasource.dataframe.to_csv(
                path_complete, sep=';', decimal = ',')
            # self.report_df.to_csv(path_report, sep=';', decimal = ',')

        except Exception:
            raise
            print('Error saving results, unable to save file: %r', path)

    def show_message(self):

        print("Running simulation for source data file: {0} from: \
              {1} to {2}".format(
            self.parameters['simulation']['filename'],
            self.parameters['simulation']['first_date'],
            self.parameters['simulation']['last_date']))
        print("Model: {0}".format(
            self.parameters['model']['name']))
        print("Simulation: {0} ; Benchmark: {1} ; FastMode: {2}".format(
            self.parameters['simulation']['simulation'],
            self.parameters['simulation']['benchmark'],
            self.parameters['simulation']['fastmode']))

        print("Site: {0} @ Lat: {1:.2f}º, Long: {2:.2f}º, Alt: {3} m".format(
            self.site.name, self.site.latitude,
            self.site.longitude, self.site.altitude))

        print("Loops:", self.solarfield.total_loops,
              'SCA/loop:', self.parameters['loop']['scas'],
              'HCE/SCA:', self.parameters['loop']['hces'])
        print("SCA model:", self.parameters['SCA']['Name'])
        print("HCE model:", self.parameters['HCE']['Name'])
        if self.parameters['HTF']['source'] == 'table':
            print("HTF form table:", self.parameters['HTF']['name'])
        elif self.parameters['HTF']['source'] == 'CoolProp':
            print("HTF form CoolProp:", self.parameters['HTF']['CoolPropID'])
        print("------------------------------------------------------------")


class LoopSimulation(object):
    """
    Definimos la clase simulacion para representar las diferentes
```

```
    pruebas que lancemos, variando el archivo TMY, la configuracion del
    site, la planta, el modo de operacion o el modelo empleado.
    """

    def __init__(self, settings):

        self.tracking = True
        self.htf = None
        self.site = None
        self.datasource = None
        self.parameters = settings

        if settings['model']['name'] == 'Barbero4thOrder':
            self.model = ModelBarbero4thOrder(settings['model'])
        elif settings['model']['name'] == 'Barbero1stOrder':
            self.model = ModelBarbero1stOrder(settings['model'])
        elif settings['model']['name'] == 'BarberoSimplified':
            self.model = ModelBarberoSimplified(settings['model'])

        self.datasource = TableData(settings['simulation'])

        self.site = Site(settings['site'])

        if settings['HTF']['source'] == "CoolProp":
            if settings['HTF']['CoolPropID'] not in Fluid._COOLPROP_FLUIDS:
                print("Not CoolPropID valid")
                sys.exit()
            else:
                self.htf = FluidCoolProp(settings['HTF'])
        else:
            self.htf = FluidTabular(settings['HTF'])

        self.base_loop = BaseLoop(settings['loop'],
                                  settings['SCA'],
                                  settings['HCE'])

    def runSimulation(self):

        self.show_message()

        flag_0 = datetime.now()

        for row in self.datasource.dataframe.iterrows():

            solarpos = self.site.get_solarposition(row)

            if solarpos['zenith'][0] < 90:
                self.tracking = True
            else:
                self.tracking = False

            self.simulate_base_loop(solarpos, row)

            str_data = ("{0} Ang. Zenith: {1:.2f} DNI: {2} W/m2 " +
                        "Qm: {3:.1f}kg/s Tin: {4:.1f}K Tout: {5:1f}K")

            print(str_data.format(row[0], solarpos['zenith'][0],
                                  row[1]['DNI'], self.base_loop.act_massflow,
                                  self.base_loop.tin, self.base_loop.tout))

```

```
2519        print(self.datasource.dataframe)
2520
2521        flag_1 = datetime.now()
2522        delta_t = flag_1 - flag_0
2523        print("Total runtime: ", delta_t.total_seconds())
2524
2525        self.save_results()
2526
2527    def simulate_base_loop(self, solarpos, row):
2528
2529        values = {'tin': 573,
2530                  'pin': 1900000,
2531                  'massflow': 4}
2532        self.base_loop.initialize('values', values)
2533        HCE_var = ''
2534        SCA_var = ''
2535
2536        for c in row[1].keys():
2537            if c in self.base_loop.parameters_sca.keys():
2538                SCA_var = c
2539
2540        for c in row[1].keys():
2541            if c in self.base_loop.parameters_hce.keys():
2542                HCE_var = c
2543
2544        for s in self.base_loop.scas:
2545            if SCA_var != '':
2546                s.parameters[SCA_var] = row[1][SCA_var]
2547            aoi = s.get_aoi(solarpos)
2548            for h in s.hces:
2549                if HCE_var != '':
2550                    h.parameters[HCE_var] = row[1][HCE_var]
2551                h.set_pr_opt(solarpos)
2552                h.set_qabs(aoi, solarpos, row)
2553                h.set_tin()
2554                h.set_pin()
2555                h.tout = h.tin
2556                self.model.calc_pr(h, self.htf, row)
2557
2558        self.base_loop.tout = self.base_loop.scas[-1].hces[-1].tout
2559        self.base_loop.pout = self.base_loop.scas[-1].hces[-1].pout
2560        self.base_loop.set_loop_values_from_HCEs('actual')
2561        print('pr', self.base_loop.pr_act_massflow ,
2562              'tout', self.base_loop.tout,
2563              'massflow', self.base_loop.massflow)
2564
2565        if HCE_var +  SCA_var != '':
2566            self.datasource.dataframe.at[row[0], HCE_var + SCA_var] = row[1][HCE_var
    + SCA_var]
2567        self.datasource.dataframe.at[row[0], 'pr'] = self.base_loop.pr_act_massflow
2568        self.datasource.dataframe.at[row[0], 'tout'] = self.base_loop.tout
2569        self.datasource.dataframe.at[row[0], 'pout'] = self.base_loop.pout
2570        self.datasource.dataframe.at[row[0], 'Z'] = solarpos['zenith'][0]
2571        self.datasource.dataframe.at[row[0], 'E'] = solarpos['elevation'][0]
2572        self.datasource.dataframe.at[row[0], 'aoi'] = aoi
2573
2574    def save_results(self):
2575
2576
2577        try:
```

```
            initialdir = "./simulations_outputs/"
            prefix = datetime.today().strftime("%Y%m%d %H%M%S")
            filename = "Loop Simulation"
            sufix = ".csv"

            path = initialdir + prefix + filename + sufix

            self.datasource.dataframe.to_csv(path, sep=';', decimal = ',')

        except Exception:
            raise
            print('Error saving results, unable to save file: %r', path)


    def show_message(self):

        print("Running simulation for source data file: {0}".format(
            self.parameters['simulation']['filename']))

        print("Site: {0} @ Lat: {1:.2f}º, Long: {2:.2f}º, Alt: {3} m".format(
            self.site.name, self.site.latitude,
            self.site.longitude, self.site.altitude))

        print('SCA/loop:', self.parameters['loop']['scas'],
              'HCE/SCA:', self.parameters['loop']['hces'])
        print("SCA model:", self.parameters['SCA']['Name'])
        print("HCE model:", self.parameters['HCE']['Name'])
        print("HTF:", self.parameters['HTF']['name'])
        print("------------------------------------------------")


class Fluid:

    _T_REF = 285.856  # Kelvin, T_REF= 12.706 Celsius Degrees
    _COOLPROP_FLUIDS = ['Water', 'INCOMP::TVP1', 'INCOMP::S800']

    def test_fluid(self, tmax, tmin, p):

        data = []

        for tt in range(int(round(tmax)), int(round(tmin)), -5):
            data.append({'T': tt,
                         'P': p,
                         'cp': self.get_specific_heat(tt, p),
                         'rho': self.get_density(tt, p),
                         'mu': self.get_dynamic_viscosity(tt, p),
                         'kt': self.get_thermal_conductivity(tt, p),
                         'H': self.get_enthalpy(tt, p),
                         'T-H': self.get_temperature(self.get_enthalpy(tt, p), p)})
        df = pd.DataFrame(data)
        print(round(df, 6))

    def get_specific_heat(self, p, t):
        pass

    def get_density(self, p, t):
        pass

    def get_thermal_conductivity(self, p, t):
        pass
```

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

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

```
2758
2759        if t > self.tmax:
2760            t = self.tmax
2761
2762        CP.set_reference_state(self.coolpropID,'ASHRAE')
2763        hin = PropsSI('H', 'T', t,  'P', p, self.coolpropID)
2764        try:
2765            temperature = PropsSI('T', 'H', hin  + q/mf, 'P', p, self.coolpropID)
2766        except:
2767            temperature = self.tmax
2768        CP.set_reference_state(self.coolpropID, 'DEF')
2769
2770        return temperature
2771
2772 class FluidTabular(Fluid):
2773
2774     def __init__(self, settings=None):
2775
2776         self.name = settings['name']
2777         self.cp = settings['cp']
2778         self.rho = settings['rho']
2779         self.mu = settings['mu']
2780         self.kt = settings['kt']
2781         self.h = settings['h']
2782         self.t = settings['t']
2783         self.tmax = settings['tmax']
2784         self.tmin = settings['tmin']
2785
2786
2787     def get_density(self, t, p):
2788
2789         # Dowtherm A.pdf, 2.2 Single Phase Liquid Properties. pg. 8.
2790
2791         poly = np.polynomial.polynomial.Polynomial(self.rho)
2792
2793         return poly(t)
2794
2795     def get_dynamic_viscosity(self, t, p):
2796
2797         poly = np.polynomial.polynomial.Polynomial(self.mu)
2798
2799         mu = poly(t)
2800
2801         return mu
2802
2803     def get_specific_heat(self, t, p):
2804
2805         poly = np.polynomial.polynomial.Polynomial(self.cp)
2806
2807         return poly(t)
2808
2809     def get_thermal_conductivity(self, t, p):
2810         ''' Saturated Fluid conductivity at temperature t '''
2811
2812         poly = np.polynomial.polynomial.Polynomial(self.kt)
2813
2814         return poly(t)
2815
2816     def get_enthalpy(self, t, p):
2817
```

```
        poly = np.polynomial.polynomial.Polynomial(self.h)

        return poly(t)

    def get_delta_enthalpy(self, t1, t2, p1, p2):

        cp0, cp1, cp2, cp3, cp4, cp5, cp6, cp7, cp8 = tuple(self.cp)

        h = (
            (cp0 * t2 + cp1 * t2**2 / 2 + cp2 * t2**3 / 3 +
             cp3 * t2**4 / 4 + cp4 * t2**5 / 5 + cp5 * t2**6 / 6 +
             cp6 * t2**7 / 7 + cp7 * t2**8 / 8 + cp8 * t2**9 / 9)
            -
            (cp0 * t1 + cp1 * t1**2 / 2 + cp2 * t1**3 / 3 +
             cp3 * t1**4 / 4 + cp4 * t1**5 / 5 + cp5 * t1**6 / 6 +
             cp6 * t1**7 / 7 + cp7 * t1**8 / 8 + cp8 * t1**9 / 9))

        return h

    def get_temperature(self, h, p):

        poly = np.polynomial.polynomial.Polynomial(self.t)

        return poly(h)

    def get_temperature_by_integration(self, tin, h, mf=None, p=None):


        cp0, cp1, cp2, cp3, cp4, cp5, cp6, cp7, cp8 = tuple(self.cp)

        a0 = (h/mf + cp0 * tin + cp1 * tin**2 / 2 + cp2 * tin**3 / 3 +
              cp3 * tin**4 / 4 + cp4 * tin**5 / 5 + cp5 * tin**6 / 6 +
              cp6 * tin**7 / 7 + cp7 * tin**8 / 8 + cp8 * tin**9 / 9)

        factors = [a0, -cp0, -cp1 / 2, -cp2 / 3, -cp3 / 4, -cp4 / 5, -cp5 / 6,
                   -cp6 / 7, -cp7 / 8, -cp8 / 9]

        poly = np.polynomial.polynomial.Polynomial(factors)
        roots = poly.roots()

        tout_bigger = []
        tout_smaller = []

        for r in roots:
            if r.imag == 0.0:
                if r.real >= tin:
                    tout_bigger.append(r.real)
                else:
                    tout_smaller.append(r.real)
        if h > 0:
            tout = min(tout_bigger)
        elif h<0:
            tout = max(tout_smaller)
        else:
            tout = tin

        return tout


class Weather(object):
```

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

```
        for c in self.dataframe.columns:
            if (c == 'DryBulb') or (c == 'DewPoint'): # From Celsius Degrees to K
                self.dataframe[c] *= 0.1
                self.dataframe[c] += 273.15
            if c=='Pressure': # from mbar to Pa
                self.dataframe[c] *= 1e2

    def filter_columns(self):

        needed_columns = ['DNI', 'DryBulb', 'DewPoint', 'Wspd', 'Pressure']
        columns_to_drop = []
        for c in  self.dataframe.columns:
            if c not in needed_columns:
                columns_to_drop.append(c)
        self.dataframe.drop(columns = columns_to_drop, inplace = True)

    def site_to_dict(self):
        """
        pvlib.iotools realiza modificaciones en los nombres de las columnas.
        """

        return {"name": 'nombre_site',
                "latitude": self.site['latitude'],
                "longitude": self.site['longitude'],
                "altitude": self.site['altitude']}

class FieldData(object):

    def __init__(self, settings, tags = None):
        self.filename = settings['filename']
        self.filepath = settings['filepath']
        self.file = self.filepath + self.filename
        self.first_date = pd.to_datetime(settings['first_date'])
        self.last_date = pd.to_datetime(settings['last_date'])
        self.tags = tags
        self.dataframe = None

        self.openFieldDataFile(self.file)
        self.rename_columns()
        self.change_units()


    def openFieldDataFile(self, path = None):

        '''
        fielddata
        '''

        rows_list =[]
        index_count = 1  # Skip fist row in skiprows: it has got columns names
        #dateparse = lambda x: pd.datetime.strptime(x, '%YYYY/%m/%d %H:%M')
        try:
            if path is None:
                root = Tk()
                root.withdraw()
                path = askopenfilename(initialdir = ".fielddata_files/",
                                       title = "choose your file",
                                       filetypes = (("csv files","*.csv"),
                                                    ("all files","*.*")))
                root.update()
```

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

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

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