```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']

    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
        tro = tf + qabs * pr / urec

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

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

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

        #  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
```

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

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

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

```
            self.tin, q, self.sca.loop.massflow, self.pin)


    def set_pout(self, htf):

        '''
        TO-DO: CÁLCULLO PERDIDA DE CARGA:

        Ec. Colebrook-White para el cálculo del factor de fricción de Darcy
        re_turbulent = 2300

        k = self.parameters['Inner surface roughness']
        D = self.parameters['Absorber tube inner diameter']
        re = htf.get_Reynolds(D, self.tin, self.pin, self.sca.loop.massflow)

        if re < re_turbulent:
            darcy_factor = 64 / re

        else:
            # a = (k /  D) / 3.7
            a = k / 3.7
            b = 2.51 / re
            x = 1

            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']))
```

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

```
                                    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'])
```

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

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

```
        self.status = 'focused'
        self.tracking_angle = 0.0
        self.parameters = dict(settings)


    def get_solar_fraction(self):

        if self.status == 'defocused':
            solarfraction = 0.0
        elif self.status == 'focused':

            #  Cleanliness two times because it affects mirror and envelope
            solarfraction = (self.parameters['Geom.Accuracy'] *
                             self.parameters['Track Twist'] *
                             self.parameters['Cleanliness'] *
                             self.parameters['Cleanliness'] *
                             self.parameters['Factor'] *
                             self.parameters['Availability'])
        else:
            solarfraction = 1.0

        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['IAM Coefficient F0']
            F1 = self.parameters['IAM Coefficient F1']
            F2 = self.parameters['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.loop.parameters['Tracking Type'] == 1:  # N-S single axis
            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
```

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

```
                    h.parameters['Length'] *
                    h.parameters['Absorber tube outer diameter'])
                qlost_list.append(
                    h.qlost *
                    np.pi *
                    h.parameters['Length'] *
                    h.parameters['Absorber tube outer diameter'])
                pr_opt_list.append(h.pr_opt)

        self.pr = np.mean(pr_list)
        self.qabs = np.sum(qabs_list)
        self.qlost_brackets = np.sum(qlost_brackets_list)
        self.qlost = np.sum(qlost_list)
        self.pr_opt = np.mean(pr_opt_list)

    def load_from_base_loop(self, base_loop):

        self.massflow = base_loop.massflow
        self.tin = base_loop.tin
        self.pin = base_loop.pin
        self.tout = base_loop.tout
        self.pout = base_loop.pout
        self.pr = base_loop.pr
        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)
```

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

```
            _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
```

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

```
        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
    '''

```

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

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

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

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

```
            np.sum([sb.massflow * htf.get_enthalpy(sb.tout, sb.pout)
                    for sb in self.subfields]) / self.massflow, self.pout)
        self.qlost = np.sum([sb.qlost for sb in self.subfields])
        self.qabs = np.sum([sb.qabs for sb in self.subfields])
        self.qlost_brackets = np.sum(
            [sb.qlost_brackets for sb in self.subfields])

    def set_massflow(self):

        self.massflow = np.sum([sb.massflow for sb in self.subfields])
        self.req_massflow = np.sum([sb.req_massflow for sb in self.subfields])

    def set_req_massflow(self):

        self.req_massflow = np.sum([sb.req_massflow for sb in self.subfields])

    def set_wasted_power(self):

        self.wasted_power = np.sum([sb.wasted_power for sb in self.subfields])

    def set_pr_req_massflow(self):

        self.pr_req_massflow = np.sum(
            [sb.pr_req_massflow * sb.req_massflow for sb in self.subfields]) \
            / self.req_massflow

    def set_pr_act_massflow(self):

         self.pr_act_massflow = np.sum(
            [sb.pr_act_massflow * sb.act_massflow for sb in self.subfields]) \
            / self.act_massflow

    def set_pout(self):

        self.pout = np.sum(
            [sb.pout * sb.massflow for sb in self.subfields]) \
            / self.massflow

    def set_tout(self, htf):
        """
        Calculates HTF output temperature throughout the solar plant as a
        weighted average based on the enthalpy of the mass flow in  each
        subfield of the solar field
        """
        self.tout = htf.get_temperature(
            np.sum([sb.massflow * htf.get_enthalpy(sb.tout, sb.pout) for sb in
             self.subfields]) / self.massflow, self.pout)

    def set_act_tout(self, htf):
        """
        Calculates HTF output temperature throughout the solar plant as a
        weighted average based on the enthalpy of the mass flow in each
        loop of the solar field
        """
        self.act_tout = htf.get_temperature(
            np.sum([sb.act_massflow *
                    tf.get_enthalpy(sb.act_tout, sb.act_pout) for sb in
             self.subfields]) / self.act_massflow, self.act_pout)

    def set_tin(self, htf):
```

```
        """
        Calculates HTF output temperature throughout the solar plant as a
        weighted average based on the enthalpy of the mass flow in each
        loop of the solar field
        """
        self.tin = tf.get_temperature(
            np.sum([sb.massflow *
                    tf.get_enthalpy(sb.tin, sb.pin) for sb in
             self.subfields]) / self.massflow, self.pin)

    def set_pin(self):

        self.pin = np.sum([sb.pin * sb.massflow for sb in self.subfields]) \
            / self.massflow

    def set_act_pin(self):

        self.act_pin = np.sum(
            [sb.act_pin * sb.act_massflow for sb in self.subfields]) \
            / self.massflow

    def set_thermal_power(self, htf, datatype):

        self.pwr = self.massflow * \
            htf.get_delta_enthalpy(self.tin, self.tout, self.pin, self.pout)

        #  From watts to MW
        self.pwr /= 1000000

        if datatype == 2:
            self.act_pwr = self.act_massflow * \
                htf.get_delta_enthalpy(
                    self.act_tin, self.act_tout, self.act_pin, self.act_pout)

            #  From watts to MW
            self.act_pwr /= 1000000

    def print(self):

        for sb in self.subfields:
            for l in sb.loops:
                for s in l.scas:
                    for h in s.hces:
                        print("subfield: ", sb.name,
                              "Lazo: ",l.loop_order,
                              "SCA: ", s.sca_order,
                              "HCE: ", h.hce_order,
                              "tin", "=", h.tin,
                              "tout", "=", h.tout)


class SolarFieldSimulation(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):

```

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

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

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

```
                    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):
```

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

```
                    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,
                s.get_id() + '.x.qlst': s.qlost,
                s.get_id() + '.x.qlbk': s.qlost_brackets}

            self.store_values(row, values)

            if not self.fastmode:
                for l in s.loops:
                    #  Loop data
                    values = {
                        l.get_id() + '.x.mf':  l.massflow,
                        l.get_id() + '.x.tin':  l.tin - 273.15,
                        l.get_id() + '.x.tout':  l.tout - 273.15,
                        l.get_id() + '.x.pin':  l.pin,
                        l.get_id() + '.x.pout':  l.pout,
                        l.get_id() + '.x.prth':  l.pr,
                        l.get_id() + '.x.prop':  l.pr_opt,
                        l.get_id() + '.x.qabs':  l.qabs,
                        l.get_id() + '.x.qlost':  l.qlost,
                        l.get_id() + '.x.qlbk':  l.qlost_brackets}

                    self.store_values(row, values)

    def gather_benchmark_data(self, row):

        #  Solarfield data
        self.datasource.dataframe.at[row[0], 'SF.a.mf'] = \
            self.solarfield.massflow
        self.datasource.dataframe.at[row[0], 'SF.a.tin'] = \
            self.solarfield.tin - 273.15
        self.datasource.dataframe.at[row[0], 'SF.a.tout'] = \
            self.solarfield.act_tout - 273.15
        self.datasource.dataframe.at[row[0], 'SF.a.pwr'] = \
            self.solarfield.act_pwr
        self.datasource.dataframe.at[row[0], 'SF.b.tout'] = \
            self.solarfield.tout - 273.15
        self.datasource.dataframe.at[row[0], 'SF.b.prth'] = \
```

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

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

```
                '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']))
```

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

```
2340    def runSimulation(self):
2341
2342        self.show_message()
2343
2344        flag_0 = datetime.now()
2345
2346        for row in self.datasource.dataframe.iterrows():
2347
2348            solarpos = self.site.get_solarposition(row)
2349
2350            if solarpos['zenith'][0] < 90:
2351                self.tracking = True
2352            else:
2353                self.tracking = False
2354
2355            self.simulate_base_loop(solarpos, row)
2356
2357            str_data = ("{0} Ang. Zenith: {1:.2f} DNI: {2} W/m2 " +
2358                        "Qm: {3:.1f}kg/s Tin: {4:.1f}K Tout: {5:1f}K")
2359
2360            print(str_data.format(row[0], solarpos['zenith'][0],
2361                                  row[1]['DNI'], self.base_loop.act_massflow,
2362                                  self.base_loop.tin, self.base_loop.tout))
2363
2364        print(self.datasource.dataframe)
2365
2366        flag_1 = datetime.now()
2367        delta_t = flag_1 - flag_0
2368        print("Total runtime: ", delta_t.total_seconds())
2369
2370        self.save_results()
2371
2372    def simulate_base_loop(self, solarpos, row):
2373
2374        values = {'tin': 573,
2375                  'pin': 1900000,
2376                  'massflow': 4}
2377        self.base_loop.initialize('values', values)
2378        HCE_var = ''
2379        SCA_var = ''
2380
2381        for c in row[1].keys():
2382            if c in self.base_loop.parameters_sca.keys():
2383                SCA_var = c
2384
2385        for c in row[1].keys():
2386            if c in self.base_loop.parameters_hce.keys():
2387                HCE_var = c
2388
2389        for s in self.base_loop.scas:
2390            if SCA_var != '':
2391                s.parameters[SCA_var] = row[1][SCA_var]
2392            aoi = s.get_aoi(solarpos)
2393            for h in s.hces:
2394                if HCE_var != '':
2395                    h.parameters[HCE_var] = row[1][HCE_var]
2396                h.set_pr_opt(solarpos)
2397                h.set_qabs(aoi, solarpos, row)
2398                h.set_tin()
2399                h.set_pin()
```

```
                h.tout = h.tin
                self.model.calc_pr(h, self.htf, row)

        self.base_loop.tout = self.base_loop.scas[-1].hces[-1].tout
        self.base_loop.pout = self.base_loop.scas[-1].hces[-1].pout
        self.base_loop.set_loop_values_from_HCEs('actual')
        print('pr', self.base_loop.pr_act_massflow ,
              'tout', self.base_loop.tout,
              'massflow', self.base_loop.massflow)

        if HCE_var +  SCA_var != '':
            self.datasource.dataframe.at[row[0], HCE_var + SCA_var] = row[1][HCE_var
+ SCA_var]
        self.datasource.dataframe.at[row[0], 'pr'] = self.base_loop.pr_act_massflow
        self.datasource.dataframe.at[row[0], 'tout'] = self.base_loop.tout
        self.datasource.dataframe.at[row[0], 'pout'] = self.base_loop.pout
        self.datasource.dataframe.at[row[0], 'Z'] = solarpos['zenith'][0]
        self.datasource.dataframe.at[row[0], 'E'] = solarpos['elevation'][0]
        self.datasource.dataframe.at[row[0], 'aoi'] = aoi

    def save_results(self):


        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']

```

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

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

```
        CP.set_reference_state(self.coolpropID,'ASHRAE')
        deltaH = PropsSI('H','T',t ,'P', p, self.coolpropID)
        CP.set_reference_state(self.coolpropID, 'DEF')

        return deltaH

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

        CP.set_reference_state(self.coolpropID,'ASHRAE')
        h1 = PropsSI('H','T',t1 ,'P', p1, self.coolpropID)
        h2 = PropsSI('H','T',t2 ,'P', p2, self.coolpropID)
        CP.set_reference_state(self.coolpropID, 'DEF')

        return mf * (h2-h1)

    def get_temperature(self, h, p):

        CP.set_reference_state(self.coolpropID,'ASHRAE')
        temperature = PropsSI('T', 'H', h, 'P', p, self.coolpropID)
        CP.set_reference_state(self.coolpropID, 'DEF')

        return temperature

    def get_temperature_by_integration(self, t,  q, mf = None, p = None):

        if t > self.tmax:
            t = self.tmax

        CP.set_reference_state(self.coolpropID,'ASHRAE')
        hin = PropsSI('H', 'T', t,  'P', p, self.coolpropID)
        try:
            temperature = PropsSI('T', 'H', hin  + q/mf, 'P', p, self.coolpropID)
        except:
            temperature = self.tmax
        CP.set_reference_state(self.coolpropID, 'DEF')

        return temperature

class FluidTabular(Fluid):

    def __init__(self, settings=None):

        self.name = settings['name']
        self.cp = settings['cp']
        self.rho = settings['rho']
        self.mu = settings['mu']
        self.kt = settings['kt']
        self.h = settings['h']
        self.t = settings['t']
        self.tmax = settings['tmax']
        self.tmin = settings['tmin']


    def get_density(self, t, p):

        # Dowtherm A.pdf, 2.2 Single Phase Liquid Properties. pg. 8.

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

        return poly(t)
```

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

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

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

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

```
                                    'Unable to open file: %r', self.file)

        self.dataframe.index = pd.to_datetime(self.dataframe.index,
                                              format= "%d/%m/%Y %H:%M")

    def change_units(self):

        for c in self.dataframe.columns:
            if ('.a.t' in c) or ('DryBulb' in c) or ('Dew' in c):
                self.dataframe[c] += 273.15 # From Celsius Degrees to K
            if '.a.p' in c:
                self.dataframe[c] *= 1e5 # From Bar to Pa
            if 'Pressure' in c:
                self.dataframe[c] *= 1e2 # From mBar to Pa

    def rename_columns(self):

        # Replace tags  with names as indicated in configuration file
        # (field_data_file: tags)

        rename_dict = dict(zip(self.tags.values(), self.tags.keys()))
        self.dataframe.rename(columns = rename_dict, inplace = True)


        # Remove unnecessary columns
        columns_to_drop = []
        for c in  self.dataframe.columns:
            if c not in self.tags.keys():
                columns_to_drop.append(c)
        self.dataframe.drop(columns = columns_to_drop, inplace = True)

class TableData(object):

    def __init__(self, settings):
        self.filename = settings['filename']
        self.filepath = settings['filepath']
        self.file = self.filepath + self.filename
        self.dataframe = None

        self.openDataFile(self.file)


    def openDataFile(self, path = None):

        '''
        Table ddata
        '''

        try:
            if path is None:
                root = Tk()
                root.withdraw()
                path = askopenfilename(initialdir = ".data_files/",
                                       title = "choose your file",
                                       filetypes = (("csv files","*.csv"),
                                                    ("all files","*.*")))
                root.update()
                root.destroy()
            else:
                strfilename, strext = os.path.splitext(path)
```

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

```
2999        DELTA_01 = FLAG_01 - FLAG_00
3000        print("Total runtime: ", DELTA_01.total_seconds())
```