Components :

Ressources :

- Physical parameters of the model
- Parameters of the numeric simulation dx and dt (steps for discretized space and time)
- *function Newton()*: $\begin{vmatrix} fun * fun * \mathbb{R} * [0,1] \\ f * f' * x_0 * \varepsilon \end{vmatrix} \rightarrow \mathbb{R}$

  Implement Newton method for equation resolution
- *function eff():* $\mathbb{R} * [0,1] \rightarrow [0,1]$

  Calculates efficiency for the $NUT - \varepsilon$ method

Heat_exchanger:

- Class HEX_nom

  'Implement a plate heat exchanger'
  - Attributes:
    - Aex = Total heat exchange surface (m^2)
    - $hnom$ = some constant related to the heat-transfer coefficient U
    - $Ts1$ = Primary supply temperature $(°C)$
    - $Tr1$ = Primary return temperature $(°C)$
    - $Tr2$ = Secondary return temperature $(°C)$ - *constant*
    - $Ts2$ = Secondary demand $(°C)$
    - $Ts2\_vrai$ = secondary supplied temperature
    - f_{Ts2} = $fun : T_{ext} \rightarrow Ts2$ Calculates secondary demand
    - m_dot1= Primary water flow $(kg.s^{-1})$
    - m_dot2 = Secondary water flow $(kg.s^{-1})$ - *constant*
    - $qmax$ = Maximum allowed water flow $(kg.s^{-1})$

  - Method __init__(): $\begin{vmatrix} self * \mathbb{R} * \mathbb{R} * vect\_fun * \mathbb{R} * \mathbb{R} * \mathbb{R} \\ self, Q_{NOM}, Tr2, f_{Ts2}, h_{NOM}, \Delta T_{LM-NOM}, \dot{m}_{max} \end{vmatrix}$
    - Q_{NOM} : Nominal power of the heat exchanger (W)
    - $Tr2$: Secondary return temperature (constant) $(°C)$
    - f_{Ts2} : $T_{ext} \rightarrow Ts2$ Calculates secondary demand $(°C)$
    - h_{NOM} : Constant related to the heat-transfer coefficient U
    - ΔT_{LM-NOM} : Nominal logarithmic mean temperature difference $(°C)$
    - $\dot{m}_{max}$: Maximum allowed water flow $(kg.s^{-1})$

  - Method UA(): $self$

    '*Calculates the heat-transfer coefficient U. Returns $U \times Aex$*'

    $$U = \frac{h_{NOM}}{\dot{m}_1^{-0.8} + \dot{m}_2^{-0.8}}$$

  - Method solve(): $\begin{vmatrix} self * \mathbb{R} \\ self * Ts1 \end{vmatrix}$
    - $Ts1$: Primary supply temperature $(°C)$

  'With given $Ts1$, calculates the values of $Tr1$, m_dot1 and $Ts2\_vrai$

  Uses the approximation given by *J.J.J. Chen* [1] to obtain $Tr1$ or $NUT - \varepsilon$ method if HEX fails to provide sufficient temperature.

  Raises $ValueError$ with a given value if the approximation is not valid (occurs when $Ts1$ and $Ts2$ are too distant)

Source

- Class Source

  'Implement the plate heat exchanger at the geothermal source'

  - Attributes:
    - m_dot = Geothermal water flow $(kg.s^{-1})$
    - Ts_Geo = Geothermal supply temperature $(°C)$ - *constant*
    - Tr_Geo = Geothermal return temperature $(°C)$
    - Ts_Net = Network supply temperature $(°C)$
    - Tr_Net = Network return temperature $(°C)$
    - P = Power exchanged during iteration (W)

  - Method __init__(): $\left|\begin{matrix} self\ *\mathbb{R}\ *\ \mathbb{R}*\mathbb{R}*\ \mathbb{R} \\ self, geoT,\ geoMdot, Q_{NOM}, \Delta T_{LM-NOM} \end{matrix}\right.$
    - $geoT$: Geothermal supply temperature $(°C)$
    - $geoMdot$: Geothermal water flow $(kg.s^{-1})$
    - Q_{NOM} : Nominal power of the heat exchanger (W)
    - ΔT_{LM-NOM} Nominal logarithmic mean temperature difference $(°C)$

  - Method UA(): $self \rightarrow \mathbb{R}$

    '*Calculates and return the product of heat-transfer coefficient U by heat exchange surface'*

    $$UA(t) = \frac{Q}{\Delta T_{LM}}$$

  - Method solve(): $\left|\begin{matrix} self * \mathbb{R} * \mathbb{R} \\ self * m\_dotNET * TrNET \end{matrix}\right.$
    - $TrNET$: Network return temperature $(°C)$
    - m_dotNET : Network side water flow $(kg.s^{-1})$

  ' With given m_dotNET and $TrNET$, calculates the values of Ts_Net and Tr_Geo using $NUT - \varepsilon$ method '

Pipes

- *Function step():* $\left|\begin{matrix} \mathbb{R}\ *\ \mathbb{R} \\ L * (dx_0) \end{matrix}\right. \rightarrow \mathbb{N} * \mathbb{R}$

  'for a given Length L and an objective spatial step dx_0, determine the most convenient spatial step dx for discretization, such as $dx\ < dx_0$ and $\exists\ n\ /\ L = n \times dx$. Returns n, dx'

- Class Pipe

  'A model of pipes for iteration-free thermal dynamics simulation based on [2]. Got two pipes: supply pipeline and return (cold) pipeline of same length '

  - Attributes:
    - $param$ = dictionary containing physical quantities
    - k = Constant related to the convective heat transfer coefficient of hot water to the inner wall of the pipe
    - $length$ = Length of the pipe (m)
    - $nb\_controlvolume$= number of control volumes (discretized space)
    - $pipeS\_T$= List of temperatures along the supply pipeline $(K)^*$
    - $pipeR\_T$ = List of temperatures along the return pipeline $(K)^*$

- $heat\_losses$ = Power lost during an iteration (W)

- Method __init__(): $\begin{vmatrix} self \\ self, \lambda_i, \lambda_p, \lambda_s, R_{int}, R_p, R_i, z, L \end{vmatrix} \begin{matrix} * \mathbb{R} * \mathbb{R} * \mathbb{R} * \mathbb{R} * \mathbb{R} * \mathbb{R} * \mathbb{R} * \mathbb{R} \\ \end{matrix}$
  - $\lambda_i, \lambda_p, \lambda_s$: heat conductivities of the pipe wall, insulation, and soil $(W.m^{-1}.K^{-1})$
  - R_{int}, R_p, R_i : Internal, external, external with insulation pipe radius (m)
  - z : Buried depth (m)
  - L : Length of the pipeline (m)

- Method R(): $\begin{vmatrix} self * \mathbb{R} \\ self * m_{dot} \end{vmatrix} \rightarrow \mathbb{R}$
  '*calculates and return the thermal resistance of the pipe, taking convection into consideration, as a function of mass flow only*'

- Method evolS_T (): $\begin{vmatrix} self * \mathbb{R} * \mathbb{R} \\ self * m_{dot} * T_{in} \end{vmatrix}$
  - m_{dot} : Water flow in the pipe $(kg.s^{-1})$
  - T_{in} : Upstream temperature $(°C)$

  'Calculates the evolution of temperatures in the supply pipe during on time step. Also calculates power heat losses'

- Method TS_ext (): $self \rightarrow \mathbb{R}$
  ' Returns downstream outlet temperature of the supply pipe in °C'

- And respectively methods evolR_T (): $\begin{vmatrix} self * \mathbb{R} * \mathbb{R} \\ self * m_{dot} * T_{r-in} \end{vmatrix}$
  TR_ext (): $self \rightarrow \mathbb{R}$)

Storage

- Class Buffer
  'Two layers model for water storage'
  - Attributes:
    - hot_T = Hot water storage temperature $(°C)$
    - low_T = Cold water storage temperature $(°C)$
    - hot_V = Hot water storage volume (m^3)
    - low_V = Cold water storage volume (m^3)
    - m_dot = water flow in&out of the storage: $> 0 \Leftrightarrow$ hot water delivery to network

  - Method __init__(): $\begin{vmatrix} self \\ self, hT, lT, hV, lV \end{vmatrix} \begin{matrix} * \mathbb{R} * \mathbb{R} * \mathbb{R} * \mathbb{R} * \mathbb{R} \\ \end{matrix}$
    - hT, lT, hV, lV : respectively initial $hot\_T, low\_T, hot\_V, low\_V$

  - Method intake_hot_water (): $\begin{vmatrix} self * \mathbb{R} \\ self, \ T \end{vmatrix}$
    '*knowing entrance temperature and water flow* T *and* $self.m\_dot$*, calculates the new values of* hot_V *and* hot_T*'*

- Method intake_cold_water (): $\left|\begin{matrix} self * \mathbb{R} * \mathbb{R} \\ self,\ mdot,\ T \end{matrix}\right. \rightarrow \mathbb{R}^{+}$
  '*knowing entrance temperature and water flow instruction T and $mdot$, verifies that hot storage can be discharged of $dt \times mdot$ and accordingly calculates the new values of low_V and low_T. Returns $|mdot|$, or 0 if no water was admitted*'

- Method delivery_hot_water(): $self \rightarrow \mathbb{R} * \mathbb{R}^{+}$
  '*knowing outgoing water flow $self.mdot$, calculates the new value of hot_V and returns $hot\_T,\ |mdot|$ '*

- Method delivery_cold_water (): $\left|\begin{matrix} self * \mathbb{R} \\ self,\ mdot \end{matrix}\right. \rightarrow \mathbb{R}^{+}$
  '*knowing outgoing water flow instruction $mdot$, verifies that cold water storage can be discharged of $dt \times mdot$ and accordingly calculates the new values of low_V. Returns $low\_T, |mdot|$, with $mdot = 0$ if no water was delivered.'*

→ *During an iteration, intake_cold_water or delivery_cold_water shall always be called before using respectively delivery_hot_water and intake_hot_water. (ie user has to deal first with cold side)*

Network

- Class Network
  'Implement the Network model and links all the components. Network.iteration() allows the user to simulate the network evolution during one time-step dt'
  - Attributes:
    - $substations$ = list of the substations $[HEX, Pipe]$*
    - src = Network geothermal heat exchanger $(Source)$
    - $Storage$ = Network storage buffer $(Buffer)$
    - $nb\_substations$ = number of substations in the network

    - $supplyT$ = Network supply temperature $(°C)$
    - $returnT$ = Network return temperature $(°C)$
    - $subm\_dot$ = list of water flow 'demand' at each substation $(kg.s^{-1})^{*}$
    - m_dot = Network water flow (sum of substations water flow) $(kg.s^{-1})$
    - Ts_nodes= list of downstream temperature of each supply pipe $(°C)^{*}$
    - Tr_nodes = list of downstream temperature of each return pipes $(°C)^{*}$

    - $storage\_flow$ = Storage water flow instruction; > 0 if hot water delivery to the Network $(kg.s^{-1})$
    - $Boiler\_Tinstruct$ = Supply temperature instruction given to the Boiler $(°C)$

    - P_Boiler= counts the power supplied by the boiler during one iteration (W)
    - P_Geo = counts the power supplied by geothermy during one iteration(W)
    - P_demand = Power demand during iteration (W)
    - $P\_supplied$ = Power supplied during iteration (W)

- P_losses = Power lost during iteration (through pipes wall) (W)
- $Tsupply\_default\_SS$ = list of difference between Tsupplied and Trequest at each substation $(°C)^*$ during iteration
- $maxT$ = maximum reached temperature in Network (= supply)

- $NETtype$ = string containing Network type identification
- $alreadyrun$ = True if Network.iteration() has already been used

- Method __init__(): $\begin{vmatrix} self\ *\ Source\ *\ list([HEX * Pipe])\ *\ (Buffer) \\ self, source, list\_substations,\ (storage\_buffer) \end{vmatrix}$
  - $source$: Network geothermal heat exchanger
  - $list\_substations$: list of the substations
  - $storage\_buffer$: Network storage buffer

→ For the following methods, please refer to the document describing the iteration process

- Method iter_returnside(): $self$
- Method iter_supplyside(): $self$
- Method storage_cold(): $self$
  Raises ValueError with a given value if storage_flow is greater than Network flow
- Method storage_hot(): $self$
- Method iteration(): $self$

Model-SIM

- Class Simulation
  'A class to rule them all. Allows the user to initialise the model, to run the model with instructions, to optimise the linear law for supply temperature, and to calculate the cost of an operational strategy'
  - Attributes:
    - Ta = list of outside temperature measured every hour $(°C)$. Property: access to this attribute is controlled.
    - f_Ts1 = Linear function which links Ta and Ts1_requested
    - nb_hour = duration of the simulation (hour) = len(Ta)
    - RES = Network
    - nb_SS = number of substations in the Network
    - t = number of time step during the simulation
    - E_Geo = The total amount of energy supplied by geothermia (J/dt)
    - E_boiler = The total amount of energy supplied by the boiler (J/dt)
    - $E\_default$= Difference between energy requested and energy supplied
    - P_boiler = list of the power supplied by the boiler at each time step
    - $Demand$ = Amount of energy requested
    - $Demand\_supplied$= Amount of energy supplied to the substation
    - $heat\_losses$ = Amount of energy lost through the pipe wall

- $cost\_Tdefault\_SS$= list of the total cost due to differences between requested temperature and supplied temperature for each substation.
- $cost\_constraintT$ = cost due to overheated maximum temperature
- $storage$ = True if the network possesses a storage
- $initialised$ = False while self.initialisation() has not been used
- $save\_init\_$ = backup of the initial state of the Network.

- Method __init__(): $\left|\begin{matrix} self \ * Network * \mathbb{R}^n \\ self, RES, Ta \end{matrix}\right.$
  - RES : Network
  - Ta : list of outside temperature measured every hour ($°C$)

- Method initialisation(): $\left|\begin{matrix} self \ * (\mathbb{R}^{nb\_hour}) * (\mathbb{R}^p) \\ self, (T_{instruct}) \ , \ (Storage_{instruct}) \end{matrix}\right.$
  - $T_{instruct}$: (Optional) Instructions for supplied temperature ($°C$). Must be of length = nb_hour. Default value = None
  - $Storage_{instruct}$: (Optional) Instructions for storage water intake ($kg.s^{-1}$). Must be of length > nb_hour. Default value = None

'If initialised is False, run the model with instructions (if given) and then saves the final state. However, if initialised is True, reinitialised the model with the values given in save_init_'

- Method refined_Ts1(): $self$

'Calculates the optimal linear law (f_Ts1) (function of external temperature) for Boiler T_supply instruction'

- Method simulation(): $\left|\begin{matrix} self \ * \mathbb{R}^{nb\_hour} * (\mathbb{R}^p) \\ self, T_{instruct} \ , \ (Storage_{instruct}) \end{matrix}\right.$
  - $T_{instruct}$: Instructions for supplied temperature ($°C$). Must be of length = nb_hour.
  - $Storage_{instruct}$: (Optional) Instructions for storage water intake ($kg.s^{-1}$). Must be of length > nb_hour. Default value = None

'Initialised the model and then runs the full simulation length with the instructions given. Uses Network.iteration at each time step. Calculates the cost and stores the valuable parameters in the specific attributes. Must be used only through self.objective_function'

- Method objective_function_optim(): $\left|\begin{matrix} self \ * \mathbb{R}^{nb\_hour+p} \\ self, Instructions \end{matrix}\right. \rightarrow \mathbb{R}$
  - $Instructions$: List of instructions of minimum length nb_hour. The first nb_hour value are Temperature instructions. The optional remaining values (there has to be at least nb_hour remaining values) are Storage flow instructions.

'Adapted objective_function for the optimisation method. Returns the cost'

- Method objective_function():

$$\left|\begin{matrix} self \ * \mathbb{R}^{nb_{hour}} * (\mathbb{R}^p) * (Bool) \\ self, T_{instruct} \ , \ (Storage\_instructions), (exe\_time) \end{matrix}\right. \rightarrow \mathbb{R}$$

- $T_{instruct}$: (Optional) Instructions for supplied temperature ($°C$). Must be of length = nb_hour.
- $Storage\_instructions$: (Optional) Instructions for storage water intake ($kg.s^{-1}$). Must be of length > nb_hour. Default value = None
- exe_time = Bool. If True, print execution time. Default value = False

'Tries to run Simulation. Intercept raised ValueError and returns its value instead of calculated cost (should be sufficiently dissuasive so that the proposed Instructions will not be selected by Genetic algorithm. If no exception occurs, calculates and returns the cost for the given instructions. May be modified to print valuable information about the simulation'

- Method plot(): $\begin{vmatrix} self\ *\ \mathbb{R}^{nb\_hour} * (\mathbb{R}^p) \\ self, T_{instruct}\ ,\ (Storage_{instruct}) \end{vmatrix}$
  - $T_{instruct}$: Instructions for supplied temperature ($°C$). Must be of length = nb_hour.
  - $Storage_{instruct}$: (Optional) Instructions for storage water intake ($kg.s^{-1}$). Must be of length > nb_hour. Default value = None

'Initialised the model and then runs the full simulation length with the instructions given. Uses Network.iteration at each time step. Stores the valuable parameters in the specific attributes. Plot every information it has. WARNING: no firewall against errors so the instruction must have been previously evaluated before the plot

Plotted data: *Substations temperatures (supply and return)*
Differences between requested and supplied temperature
Boiler Power
Network water flow
Storage volume and temperature
Supply pipes delays '

- Method objective_function_optim(): $\begin{vmatrix} self\ *\ (fun) * (Bool) \\ self, (\ f\_Ts1),\ (print\_time) \end{vmatrix} \rightarrow \mathbb{R}$
  - f_Ts1 : (Optional): function calculating the instruction for Tsupply based on external temperature. Default value = None
  - $print\_time$: (Optional) Bool. True if User wants to print the execution time. Default value = False

'Adapted objective_function to evaluate a given linear function (or if not furnished self.f_Ts1) as instruction for T_supply instead of a list of instructions. Calls self.objective_function(). Returns the cost'

- Method objective_function_optim(): $\begin{vmatrix} self\ *\ (fun) \\ self, (\ f\_Ts1) \end{vmatrix}$
  - f_Ts1 : (Optional): function calculating the instruction for Tsupply based on external temperature. Default value = None

'Adapted plot() function to plot a given linear function (or if not furnished self.f_Ts1) as instruction for T_supply instead of a list of instructions. Calls self.plot()'

__Main__

- *function* optim*()*: $\left|\begin{matrix}\mathbb{N} * Simulation * (dict) * (\mathbb{N}) * (\mathbb{N}) * (Bool) \\ dim, MOD, param, step, Storage\_dim, plot\end{matrix}\right. \rightarrow \mathbb{R}^{dim+Storage\_dim}$
  - dim : Integer. Must be equal to MOD.nb_hour
  - MOD : Simulation class instance
  - $param$: (Optional) dictionary containing the algorithm parameters
  - $step$: (Optional), Integer, defines a step for the value taken by the genes of the individuals. Default value = 1
  - $Storage\_dim$: (Optional), Integer. Must be equal or greater than MOD.nb_hour. To be define only if the Network has a storage buffer. Default value =None
  - $plot$ = (Optional), Bool. If true, calls MOD.plot() at the end of the optimisation.

  'refines f_Ts1 and then runs the GA using the previous result as an exemple. Returns best Instructions found'

- *function* optim_week*()*: $\left|\begin{matrix}\mathbb{N} * Simulation * Result * (dict) * (\mathbb{R}^n) * (\mathbb{N}) \\ dim, MOD, Result\_class, param, Ta\_w, step\_optim\_h\end{matrix}\right.$
  - dim : Integer. Must be equal to MOD.nb_hour
  - MOD : Simulation class instance
  - $Result\_class$: Result class instance. Stores the results and important value such as the initial_state of the Model.
  - $param$: (Optional) dictionary containing the algorithm parameters
  - Ta_w : (Optional) Outside temperature measured every hour. Week length $(°C)$
  - $step\_optim\_h$: (Optional) Number of hours between every optimisation. For exemple, if $step\_optim\_h = 12$, we optimise the operation for 24h, then the Network works 12hours and then goes a new 24hours optimisation. Default value = 12

  'Uses optim(). The results and important values are stored in Result_class. MOD.initialisation() and MOD.initialised are used to simulate the working time of the Network between optimisations'

- Class Results:
  - Attributes:
    - $state\_init$ = initial state of the Network
    - T_Boiler_optim = list of the best-found Temperatures instruction with GA
    - $Tboiler\_Ta$ = list of the best-found Temperatures instruction with refined_f_Ts1
    - Ta_week = Outside temperature during the whole week (every hour)
    - $current\_state$ = state of the Network at the beginning of the current optimisation
    - $current\_Ta$ = Outside temperatures of the current day

GA_algorithm

'Modified genetic algorithm from Ryan (Mohammad) Solgi (2020). Accepts two new arguments: $value\_step$, which allows the user to reduce the number of potential individuals; $exemple$, which allows the users to give the genetic algorithm a previous solution so that the solution returned by the algorithm will be at least equal to that solution.'

[1] J. J. J. Chen, « Comments on improvements on a replacement for the logarithmic mean », *Chemical Engineering Science*, vol. 42, n° 10, p. 2488-2489, janv. 1987, doi: 10.1016/0009-2509(87)80128-8.

[2] X. Zheng et al., « Performance analysis of three iteration-free numerical methods for fast and accurate simulation of thermal dynamics in district heating pipeline », Applied Thermal Engineering, vol. 178, p. 115622, juin 2020, doi: 10.1016/j.applthermaleng.2020.115622.