Code for the Design and Evaluation of Heat Exchangers for Complex Fluids

Mechanical Engineering Technical Report 2017/04
Ingo H. J. Jahn
School of Mechanical and Mining Engineering
The University of Queensland.
With Contributions from: Samuel Roubin

March 26, 2017

Abstract

This report describes a quasi 1-D code that can be used for the design and evaluation of heat exchangers operating with ideal fluid, ideal gases and real gases. For a given heat exchanger defined by heat exchanger type, a range of variables defining the geometry of the heat exchangers, selected heat transfer correlations and selected pressure drop correlation the exchanger performance is evaluated. Employing a fixed heat exchanger geometry allows the performance of a given heat exchanger to be evaluated as fluid boundary conditions (massflow, inlet temperature, inlet pressure) change in both heat transfer channels. In the first instance this allows change in heat exchanger performance for a given heat exchanger to be evaluated across a range of operating conditions. Furthermore this can be used to appropriately size a heat exchanger that is required to operate at a number of conditions.

Contents

1 Introduction

When working with non ideal fluids and when considering real heat exchanger designs, simply performing an energy balance across the heat exchanger or using log mean temperature difference is no longer valid. Errors are introduced due to a number of reasons, such as non-constant heat capacities, thermal conduction in the heat exchanger casing, or boiling or condensation of the heat exchanger fluid. These problems can be overcome by splitting the heat exchanger into multiple axial slices and then performing an energy balance for each slice. The assumption is that properties (e.g. heat transfer coefficients) are constant across an individual slice. As the number of slices increases, this 1-D discretisation can fully capture step changes in fluid properties (e.g. boiling or condensation) and non linear gas properties as may be experienced with refrigerants and supercritical fluids.

`HX_solver.py` is a stand-alone open-source software that automates the evaluation of heat exchanger performance. So far *shell and tube* and *micro channel* heat exchangers have been implemented, as well as corresponding heat transfer correlations for *water*, *air*, and *supercritical Carbon Dioxide*. Using a standardised input format for the core code, this list can easily be expanded to include other types of heat exchanger designs and working fluids.

1.1 Prerequisites

`HX_solver.py` is written in python and should run into most typical computational set-ups. To ensure correct operation the following packages and minimum version requirements exist.

- `python 2.7` - any standard distribution
- `numpy`
- `scipy` version `0.18.0` or above
- `CoolProp` available from `http://www.coolprop.org/`
- `matplotlib` version `2.0.0` or above

1.2 Citing this Tool

When using the tool in simulations that lead to published works, it is requested that the following works are cited:

- Jahn, I. H. J. (2017), Code for the Design and Evaluation of Exchangers for Complex Fluids, *Mechanical Engineering Technical Report 2017/04*, The University of Queensland, Australia
- Jahn, I. H. J. (2017), Code for the Design and Evaluation of Heat Exchangers Operating with Complex Fluids, *The Journal of Open Source Software*, 2017

2 Distribution and Installation

`HX_solver.py` is distributed as part of the code collection maintained by the *Turbomachinery and Power Conversion Group* at the University of Queensland. This collection is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version. This program collection is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS

FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details `http://www.gnu.org/licenses/`.
Alternatively the code is included in the Appendix.

2.1 Installation

The code is designed to be run from the command line. The `job.py` file defining the current simulation should be stored in a local working directory. The main code file `HX_solver.py` should be added to a folder included in in both the `PYTHONPATH` and `PATH` environmental variables.

If required the installation folder can be added to the environmental variables by adding the following lines to the `.bashrc` file (or equivalent terminal start-up file).

```
export HX=${HOME}/path/to/loc/dir
export PYTHONPATH=${PYTHONPATH}:{HX}
export PATH=${PATH}:${HX}
```

After editing run:
`$ source ~./bashrc`

2.2 Modifying the Code

The working version of `HX_solver.py` is located in the `$geotherm/geobin` directory available as part of the geotherm repository. If you perform modifications or improvements to the code please submit an updated version together with a short description of the changes to the authors. Once reviewed the changes will be included in future versions of the code.

3 Simulation of Heat Exchangers

3.1 Running the Tool

The code is run by creating a simulation job file (e.g. `job.py`) which is passed to the main solver. The main solver solves the discretised equations to return lists of temperatures, pressures and heat fluxes defining the operation of the heat exchanger. To run the code follow these instructions:

1. Modify an existing job file to set simulation conditions (e.g. `HX1_micro-channel.py`, see section 5 for examples). Within this file the following information is specified:
   - Fluid conditions at heat exchanger inlets.
   - Geometric and physical parameters defining the heat exchanger geometry.
   - Select appropriate correlation for evaluation of heat transfer and frictional losses.
   - Set modelling parameters
2. Run the code from the command line:
   `$ HX_solver.py --job=job.py`

The following options are available `--help` shows usage instructions; and `--noprint` – suppresses outputs in terminal and plotting of results. The resulting plots show the temperature distributions and heat fluxes along the length of the heat exchanger. The on-screen outputs summarise the performance.

3.2 Parameter Settings - Generic

Inputs to the heat exchanger simulation code are set in a separate job file, which is called by the main routine. The inputs are split into three different classes corresponding to `M` for the model parameters, including selection of correlations, `G` the heat exchanger geometry parameters, and `F` the fluid specific parameters defining the fluid type and heat exchanger boundary conditions. The input parameters are specified using the syntax `A.xx = 123` or `A.xx='string'`, where `A` identifies the class and `xx` specifies the specific variable name.

The geometry specific parameters can be set using `G.xx = ...` and as a minimum the following are required:

- `HXtype`: Defines the Heat exchanger type and associated modeling assumptions. See section 4.1 for type specific inputs. Currently the following heat exchangers are implemented:
  - `micro-channel` - Heat exchanger consisting of individual circular micro channels (see section 4.1.1).
  - `shell-tube` - Heat exchanger consisting of a shell and internal tubes. The convention is that the H channel refers to the fluid inside the tubes (see section 4.1.2).
- `HX_L` (m): Length of the heat exchanger
- `k_wall` ($\mathrm{W\,m^{-1}\,K^{-1}}$): Thermal conductivity of heat exchanger material
- `epsilonH` or `epsilonC` the roughness height for the H and C channel

The fluid specific parameters are set using `F.xx = ....` Here a H and C channel corresponding to the hot and cold fluid path are considered. As a minimum the following parameters are required :

- `fluidH` and `fluidC` ($-$): String to specify fluid type. See CoolProp documentation for supported fluids [1].
- `TH_in` (K): Inlet temperature for H channel.
- `mdotH` (kg/s): Mass flow rate for H channel.
- `PH_in` (Pa): Inlet pressure for H channel.
- `PH_out` (Pa) (optional): Outlet pressure for H channel. If specified and no friction loss correlation is specified, a linear pressure drop will be assumed.
- `TC_in` (K): Inlet temperature for C channel.
- `mdotC` (kg/s): Mass flow rate for C channel.
- `PC_in` (Pa): Inlet pressure for C channel.
- `PC_out` (Pa) (optional): Outlet pressure for C channel. If specified and no friction loss correlation is specified, a linear pressure drop will be assumed.
- `T_ext` (K) (optional): Surrounding temperature for calculation of power loss to surrounding.
- `F.T0 = [Temp List ]` (K) (optional) This can be a list with length $4 \times M.N\_cell$ to specify a previous solution to accelerate the iterative solver.

The model specific parameters are set using `M.xx = ....` The following are required:

- `N_cell` ($-$): Number of cells used for spatial discretization
- `flag_axial` (`[0/1]`): switch to select if thermal conduction in the axial direction is to be included
- `Nu_CorrelationH` (`[x]`): select heat transfer correlation for H channel. See section 4.2 for implemented options.
- `Nu_CorrelationC` (`[x]`) (optional): select heat transfer correlation for C channel. See section 4.2 for implemented options. Will default to $CorrelationH$ if not specified.
- `f_CorrelationH` (`[1/2/3]`): select modelling approach for pressure loss due to friction in H channel.
  - `1` code automatically switches between a laminar and turbulent friction factor at $Re = 2300$.
  - `2` laminar - friction factor is calculated as $f = \frac{64}{Re}$.
  - `3` turbulent - friction factor is calculated using Haaland's formula.

  See section 4.3 for further details.
- `f_CorrelationC` (`[x]`) (optional): select modelling approach for pressure loss due to friction in C channel. Will default to $f_CorrelationH$ if not specified.
- `H_dP_poly` ($-$): list of coefficients defining pressure drop in H channel. See section 4.3 for further details.
- `C_dP_poly` ($-$): list of coefficients defining pressure drop in C channel.
- `otpim` ($-$) (optional): A string specifying the non-linear equation solver. Default is `root:hybr`. See section 3.4 for details.

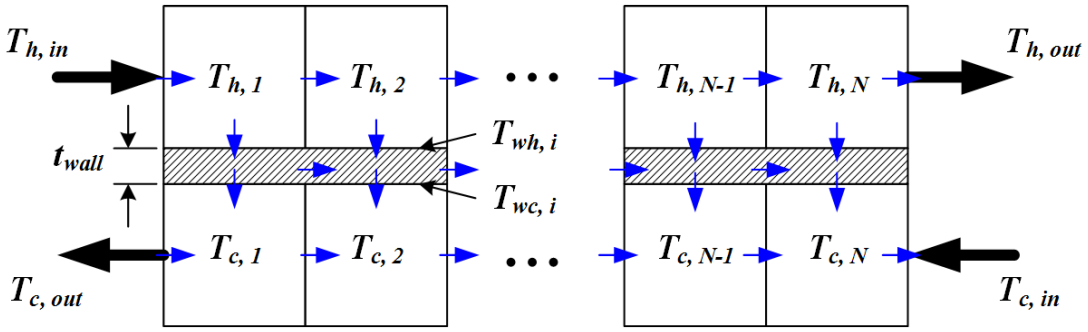


Figure 1: Modeling concept for the one-dimensional heat exchanger simulation code

3.3 Modelling Approach

The heat exchanger is modelled using three parallel one-dimensional channels as shown in Fig. 1. The first and third channel are the fluid channels and the second channel corresponds to the material separating the channels. For modelling of axial conduction this may also include casing material. During the solution process the balance of thermal energy balance is solved in the axial direction and the three channels are coupled by matching heat fluxes and temperatures at the respective interfaces. Effectively in the axial direction, within each fluid channel, a heat conduction and convection equation is solved and within the metal, a conduction equation is solved. At the same time for each set of axial slices (cell) a set of one-dimensional heat transfer equations is solved to find the heat exchanged between the two fluids and the dividing wall based on the local conditions and assuming constant properties within each cell.

This modelling approach is acceptable, as the heat exchanger performance is dominated by cross channel energy exchange and as properties vary comparatively slowly in the axial direction. Thermodynamic boundary conditions (P_in, T_in) are set on the faces of the first or last cell of the respective channel.

The following modelling assumptions are applied:

- The heat transfer in multi-channel heat exchangers can be modelled by bundling all fluid flow into two channels. The resulting heat transfer is modelled using an effective heat transfer area between the two channels and appropriate Nusselt number correlations for each fluid-solid interface to capture the effects of actual channel geometry.
- The heat transfer in the axial direction within the wall can be modelled using an effective wall area and based on local gradient in mean wall temperature T_w. Only considering the mean temperature gradient is a plausible approach as the temperature field is a superposition of axial and cross-sectional heat transfer. As temperature gradients in the channel to channel direction are much bigger than axial temperature gradients, the variation in cross-sectional heat transfer with axial position is secondary.
- The outside boundary of the heat exchanger can be modelled as adiabatic.

The solver solves for the fluid temperature in the H and C channel (T_h, T_c) and the wall temperatures in the H and C channel (T_{wh}, T_{wc}). To solve the energy balance the heat fluxes from Fig. 2 are solved as follows:

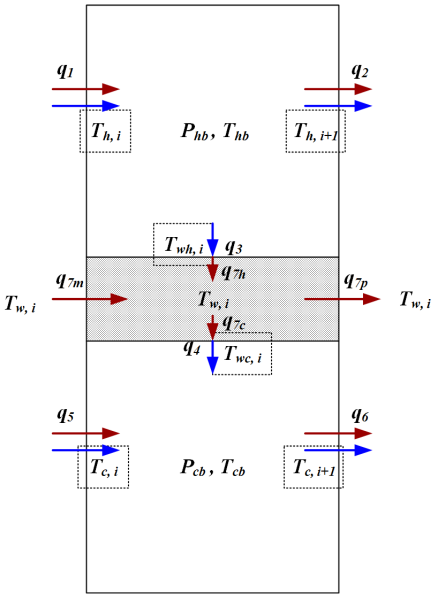


Figure 2: Energy flows considered during simulation

Heat fluxes in the H channel is given by

$$
\begin{aligned}
q_1 &= q_{1,\,cond} + q_{1,\,conv} && (3.1)\\
q_{1,\,cond} &= -\frac{\mathbf{k_{1h}}\, A_h}{\frac{L}{N_{cell}}} \left(\frac{T_{h,\,i} + T_{h,\,i+1}}{2} - \frac{T_{h,\,i-1} + T_{h,\,i}}{2} \right) \\
q_{1,\,conv} &= \mathbf{h_{h1}}\, \dot{m}_h \\
q_2 &= q_{2,\,cond} + q_{2,\,conv} && (3.2)\\
q_{2,\,cond} &= -\frac{\mathbf{k_{h2}}\, A_h}{\frac{L}{N_{cell}}} \left(\frac{T_{h,\,i+1} + T_{h,\,i+2}}{2} - \frac{T_{h,\,i} + T_{h,\,i+1}}{2} \right) \\
q_{2,\,conv} &= \mathbf{h_{h2}}\, \dot{m}_h
\end{aligned}
$$

where $\mathbf{k_{h1}}$ and $\mathbf{h_{h1}}$ and $\mathbf{k_{h2}}$ and $\mathbf{h_{h2}}$ are function calls to the CoolProp library to establish fluid thermal conductivity an enthalpy at left and right interfaces respectively.

The heat flux to the dividing wall is calculated as

$$
\begin{aligned}
q3 &= h_h\, \frac{A_{h,\,eff}}{N_{cell}} \left(\frac{T_{h,\,i} + T_{h,\,i+1}}{2} - T_{wh,\,i} \right) && (3.3)\\
h_h &= \frac{\mathbf{Nu_h}\, \mathbf{k_h}}{L_c}
\end{aligned}
$$

where A_{eff} is the effective heat transfer area for the heat exchanger, $\mathbf{k_h}$ is the fluid thermal conductivity based local bulk properties, L_c is the characteristic length and $\mathbf{Nu_h}$ is a function to evaluate the local Nusselt number. See sections 4.1 and 4.2 for selecting appropriate values L_c and an appropriate heat transfer correlation.

The heat transfer in the C channel is solved in an analogous way.

Heat conduction in the axial direction in the diving wall, which also accounts external material

(e.g. heat exchanger casing) is calculated as

$$q_{7m} = -\frac{k_{wall}\, A_{wall}}{\frac{L}{N_{cell}}} \left(\frac{T_{wh,\,i} + T_{hc,\,i}}{2} - \frac{T_{wh,\,i-1} + T_{wh,\,i-1}}{2} \right) \tag{3.4}$$

$$q_{7p} = -\frac{k_{wall}\, A_{wall}}{\frac{L}{N_{cell}}} \left(\frac{T_{wh,\,i+1} + T_{hc,\,i+1}}{2} - \frac{T_{wh,\,i} + T_{wh,\,i}}{2} \right) \tag{3.5}$$

and related to the cross wall heat fluxes as follow

$$q_{7h} = \frac{k_{wall}\, A_{h,\,eff}}{t_{wall}\, N_{cell}} \left(T_{wh,\,i} - T_{wc,\,i}\right) - 0.5 \left(-q_{7p} + q_{7m}\right) \tag{3.6}$$

$$q_{7c} = \frac{k_{wall}\, A_{h,\,eff}}{t_{wall}\, N_{cell}} \left(T_{wh,\,i} - T_{wc,\,i}\right) + 0.5 \left(-q_{7p} + q_{7m}\right). \tag{3.7}$$

Effectively this splits any mismatch in axial heat flux equally between the heat flow from channel H and flow to channel C.

The actual temperatures are solved by using the above relationships to develop $4 \times N$ non-linear energy balance equations:

$$\begin{aligned} 0 &= q_{1,\,i} - q_{2,\,i} - q_{3,\,i} \\ 0 &= q_{3,\,i} - q_{7h,\,i} \\ 0 &= q_{4,\,i} - q_{7c,\,i} \\ 0 &= q_{4,\,i} + q_{5,\,i} - q_{6,\,i} \\ &\quad \text{for } i = 1, \ldots N. \end{aligned}$$

These non-linear simultaneous equations can be solved using a selection of iterative equation solver from the *scipy* distribution. The possible solver options are listed in section 3.4 Such a solver will find a set of fluid (T_h, T_c) and wall temperatures (T_{wh}, T_{wc}) that fulfil the above equality constraints.

3.4 Options for Non-linear Equation Solver

The code uses the non-linear equations solvers available in the *scipy.optimize* module. The reader is recommended to check the *scipy* documentation for further details on the solvers. The following solvers can be selected by the user.

`root:hybr` (default): Employs MINPACKs hybrd and hybrj routines (modified Powell method) to find roots;

`root:lm`: Solves for least squares with Levenberg-Marquardt;

`root:Newton-CG`: minimises the function using the Newton-CG algorithm;

`fsolve`: Uses the function `fsolve`; and

`root:df-sane`: employs the DF-SANE method.

4 Implemented Models

4.1 Heat Exchanger Geometry Models

Modelling of the heat exchangers using the approach outlined previously requires a number of geometry specific properties. These are:

- `HX_L` (m): Length of the heat exchanger
- `Area` (m^2): Effective heat transfer area, A_{eff} that is used to calculate convective heat transfer between channels.
- `AH` (m^2): Total flow area, A_h in H channel.
- `AC` (m^2: Total flow area, A_c in C channel.
- `Lc_H` (m): Characteristic length for H channel.
- `Lc_C` (m): Characteristic length for C channel.
- `Dh_H` (m): Hydraulic diameter for H channel.
- `Dh_C` (m): Hydraulic diameter for C channel.
- `t_wall` (m): Effective thickness of wall separating channels C and H.
- `A_wall` (m^2): Effective thermal conduction area within wall in axial direction.
- `L_wall` (m): Effective length of wall.
- `k_wall` ($W\,m^{-1}\,K^{-1}$): Thermal conductivity of heat exchanger material

The following sections explain how these different parameters are implemented for the heat exchanger types that are currently supported by the code.

4.1.1 Micro Channel Heat Exchanger

Micro-channel heat exchangers, such as Printed Circuit Heat Exchangers can be modelled by approximating their geometry to a matrix of parallel tubes as shown in Fig. 3. The micro channels are modelled as circular tubes with separation t_1 and t_2 respectively. The hot and cold fluid passes through alternating horizontal rows of tubes. To model this type of heat exchanger set `G.HXtype = 'mirco-channel'` and define the following extra variables in the input `job` file:

- `N_T` ($-$): Number of tubes.
- `d` (m): Internal diameter of tubes
- `D` (m): External diameter of tubes.
- `DD` (m): Internal diameter of shell.
- `t_casing` (m: Thickness of shell.
- `d_tube` (m: Equivalent diameter of passages.

For the micro channel heat exchanger the derived parameters are calculated as follows:

- `Area` (m^2):

$$A_{eff,\,1} = N_C\,(d_{tube} + t_2) \times (2\,N_R - 1) \times HX_L \quad \text{(default)} \tag{4.1a}$$

$$A_{eff,\,2} = N_C\,N_R\,d_{tube}\,\pi HX_L \tag{4.1b}$$

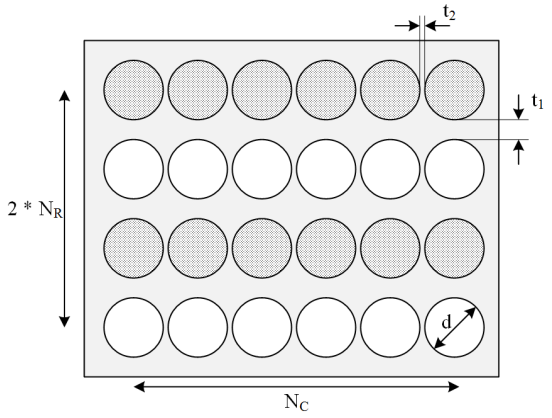


Figure 3: Geometry definition for micro channel heat exchanger

- `AH` (m^2):

$$A_h = N_R\, N_C\, \frac{d_{tube}^2}{4}\, \pi \tag{4.2}$$

- `AC` (m^2:

$$A_c = N_R\, N_C\, \frac{d_{tube}^2}{4}\, \pi \tag{4.3}$$

- `LH` (m):

$$L_h = d_{tube} \tag{4.4}$$

- `LC` (m):

$$L_c = d_{tube} \tag{4.5}$$

- `t_wall` (m):

$$t_{wall} = t_1 \tag{4.6}$$

- `A_wall` (m^2):

$$\begin{aligned} A_{wall} \quad = \quad & (N_C\, d_{tube} + (N_C - 1)\; t_2 + 2\, t_{casing}) && (4.7) \\ & \times (2\, N_R\, d_{tube} + (2\, N_R - 1)\; t_1 + 2\, t_{casing}) - A_h - A_c && (4.8) \end{aligned}$$

- `L_wall` (m):

$$t_{wall} = HX_L \tag{4.9}$$

Of the above, the most important parameter is `Area` as this defines the effective heat transfer area. For the micro channel heat exchanger specified in Fig. 3, two areas are plausible as shown in Eqn. (4.1). The first is the area of the horizontal surface separating the layers of hot and cold channels, $A_{eff,\,1}$ and is built on the assumptions that the horizontal inter-channel surface does not significantly contribute to heat exchange. The second is the surface area of the channels, $A_{eff,\,2}$. It is expected that the former is relevant for arrangements where the heat transfer is dominated by the material resistance and the latter for arrangements where fluid to wall heat transfer dominates. Comparison to experimental data for printed circuit heat exchangers (see section 5.1) showed that Eqn. (4.1a) yields better agreement to experimental data. Thus Eqn. (4.1a) has been implemented as the default method to calculate effective area.

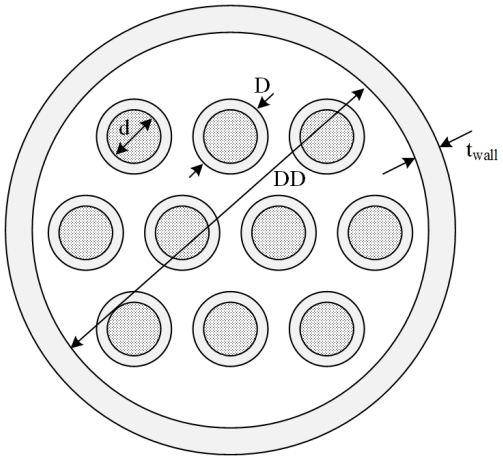


Figure 4: Geometry definition for shell and tube heat exchanger

4.1.2 Shell and Tube Heat Exchanger

Shell and tube heat exchangers are a common type of heat exchanger commonly employed. Here a number of small diameter tubes are arranged inside a larger shell as shown in Fig. 4. Within this code the convention is that the H channel is inside the tubes and the shell side is the C channel. To model this type of heat exchanger set `G.HXtype = 'shell-tube'` and define the following extra variables in the input `job` file:

- `N_T` ($-$): Number of tubes.
- `d` (m): Internal diameter of tubes
- `D` (m): External diameter of tubes.
- `DD` (m): Internal diameter of shell.
- `t_casing` (m): Thickness of shell.

For the tube and shell heat exchanger the derived parameters are calculated as follows:

- `Area` (m^2):

$$A_{eff} = d\,\pi\,HX_L \tag{4.10}$$

- `AH` (m^2):

$$A_h = N_T\,\frac{d_{tube}^2}{4}\,\pi \tag{4.11}$$

- `AC` (m^2:

$$A_c = \frac{DD^2}{4}\,\pi - N_T\,\frac{d_{tube}^2}{4}\,\pi \tag{4.12}$$

- `Lc_H` (m):

$$L_{cH} = d \tag{4.13}$$

- `Lc_C` (m):

$$L_{cC} = D \tag{4.14}$$

- `Dh_H` (m):

$$D_{hH} = d \tag{4.15}$$

- `Dh_C` (m):

$$D_{hC} = \frac{4\,A_c}{DD\,\pi + N_T\,D\,\pi} \tag{4.16}$$

- `t_wall` (m):

$$t_{wall} = \frac{D-d}{2} \tag{4.17}$$

- `A_wall` (m^2):

$$A_{wall} = DD\,\pi\,t_{wall} + N_T\,\frac{D^2-d^2}{4}\,\pi \tag{4.18}$$

- `L_wall` (m):

$$L_{wall} = HX_L \tag{4.19}$$

Of the above the most important parameter is `Area` as this defines the effective heat transfer area.

4.1.3 New Heat Exchangers

To add new heat exchanger types, the source code and the `Geomtry` class can be altered. Simply add a new class internal function analogous to `micro_init`. This function must return all the standard outputs listed at the beginning of this section.

4.2 Implemented Heat Transfer Correlations

Heat transfer is critically determined by empirical heat transfer correlations that take account of the local flow conditions, thermal properties, buoyancy effects and also geometry. To date the following correlations for Nusselt number, Nu have been implemented. The correlations can be selected by using the corresponding index to set `M.Nu_CorrelationH` and `M.Nu_CorrelationC`. The use of different correlations, for example to account for up and downwards flow, is supported.

1. CO_2 flow in pipes near the critical point, as developed by Yoon et al. [3]. Correlation has only been implemented for $T_b > T_{pc}$

$$Nu_b = 0.14\,Re_b^{0.69}\,Pr_b^{0.66} \tag{4.20}$$

   $_b$ and $_w$ refer to bulk and wall properties respectively.

2. CO_2 flow in horizontal pipes, as developed by Liao et al. [2].

$$Nu_b = 0.124\,Re_b^{0.8}\,Pr_b^{0.4}\left(\frac{Gr}{Re_b^2}\right)^{0.203}\left(\frac{\rho_w}{\rho_b}\right)^{0.842}\left(\frac{\bar{c}_p}{c_{p,\,b}}\right)^{0.384} \tag{4.21}$$

$$\bar{c}_p = \frac{i_w - i_b}{T_w - T_b} \tag{4.22}$$

   $_b$ and $_w$ refer to bulk and wall properties respectively.

Table 1: Coefficients for air flow across cylinders

Re	C	n
0.0001-0.004	0.437	0.0985
0.004-0.09	0.565	0.136
0.09-1.0	0.800	0.280
1-35	0.795	0.384
35-5000	0.583	0.471
5000-50 000	0.148	0.633
50 000 - 200 000	0.0208	0.814

3. CO_2 flow in vertical pipes and *upwards* flow, as developed by Liao et al. [2].

$$Nu_b = 0.354\, Re_b^{0.8}\, Pr_b^{0.4} \left(\frac{Gr_m}{Re_b^{2.7}}\right)^{0.157} \left(\frac{\rho_w}{\rho_b}\right)^{1.297} \left(\frac{\bar{c}_p}{c_{p,\,b}}\right)^{0.296} \tag{4.23}$$

   $_b$ and $_w$ refer to bulk and wall properties respectively.

4. CO_2 flow in vertical pipes and *downwards* flow, as developed by Liao et al. [2].

$$Nu_b = 0.643\, Re_b^{0.8}\, Pr_b^{0.4} \left(\frac{Gr_m}{Re_b^{2.7}}\right)^{0.186} \left(\frac{\rho_w}{\rho_b}\right)^{2.154} \left(\frac{\bar{c}_p}{c_{p,\,b}}\right)^{0.751} \tag{4.24}$$

   $_b$ and $_w$ refer to bulk and wall properties respectively.

5. Fluid flow inside of circular pipes, as developed by Boelter [4].

$$Nu = 0.23\, Re\, Pr^n \tag{4.25}$$
$$\text{with} \quad \begin{cases} n = 0.3 & \text{for heating of fluid} \\ n = 0.4 & \text{for cooling of fluid} \end{cases}$$

6. Fluid flow on the shell side of shell and tube heat exchangers, as per the paper by Xie et. al. [7]

$$Nu = e^{C+m\,\log RE}\, Pr^{\frac{1}{3}} \tag{4.26}$$
$$\text{with} \quad C = 0.16442; \qquad m = 0.65582$$

7. Tubes in air cross flow with characteristic length equal to tube diameter. For this case two correlations are implemented to differentiate between natural and forced convection [5].

   Natural Convection: $0 < Re < 0.0001$

$$Nu = C\,(GrPr)^n \tag{4.27}$$
$$\text{with} \quad \begin{cases} \text{for } Gr\,Pr < 1\times 10^9 & C = 0.053; \quad n = \frac{1}{4} \\ \text{for } Gr\,Pr > 1\times 10^9 & C = 0.126; \quad n = \frac{1}{3} \end{cases}$$

   Force Convection: $Re > 0.0001$

$$Nu = C\, Re \tag{4.28}$$

   with coefficients listed in Tab. 1.

4.2.1 New Heat Transfer Correlations

To add new heat transfer correlations, extend the function `calc_Nu` and add new Correlation options that can be specified. The function must return a local value of Nusselt number.

4.3 Implemented Heat Exchanger Pressure Drop Options

A number of modelling approaches have been implemented to model pressure change along the heat exchanger channels. These can be categorised in two groups. In the first a friction factor is calculated, which allows calculation of the pressure drop based on local condition (changing pressure and temperature) along the respective heat exchanger channel. The friction factor, f is converted to a pressure drop from one cell to the next using the relationship:

$$\Delta P = \rho \left(\frac{\Delta L}{D_H} f \frac{V^2}{2} \right), \tag{4.29}$$

where ΔL is the length of a cell, D_H is the hydraulic diameter of the channel, and V is the channel mean velocity.

In the second, pressures are specified at the inlet and outlet of the heat exchanger and a linear pressure drop is assumed within the heat exchanger. The outlet pressure can either be specified directly by the user, by setting `PH_out` or `PC_out`, or alternatively by specifying a list containing coefficients for a pressure drop vs mass flow rate polynomial. When a list of polynomial coefficents is defined, the pressure drop and downstream pressure are calculated by:

$$\Delta P = \sum_{i=1}^{N} a_i \dot{m}^i, \tag{4.30}$$

$$P_{out} = P_{in} - \Delta P, \tag{4.31}$$

where a_i are the polynomial coefficients defined in the lists `H_dP_poly` and `C_dP_poly` for the H and C channel respectively. When using the polynomial in either channel the friction correlation option 0 has to be selected and a linear pressure between P_{in} and P_{out} is modelled.

The following options have been implemented for the calculation of friction factor f:

1. No friction. The model will either assume a constant pressure if only inlet pressure, P_{in} is supplied, or apply a linear pressure drop if both inlet and outlet pressure are supplied.

$$f = 0 \tag{4.32}$$

2. This option automatically switches between the relationships of correlation 3 and 4 based on channel Reynolds number. For $Re \leq 2300$ correlation 3 is used, for $Re > 2300$ correlation 4 is used.
3. Laminar flow

$$f = \frac{64}{Re}. \tag{4.33}$$

4. Turbulent flow using Haaland's formula

$$\frac{1}{f^{\frac{1}{2}}} = -1.8 \log \left(\left(\frac{\frac{\epsilon}{D_H}}{3.7} \right)^1 .11 + \frac{6.9}{Re} \right), \tag{4.34}$$

where ϵ is the pipe roughness height.

4.3.1 New Friction Factor Correlations

To add new pressure loss correlations the function `calc_friction` can be extend to include new Correlation options. The function must return a local value of friction factor, f.

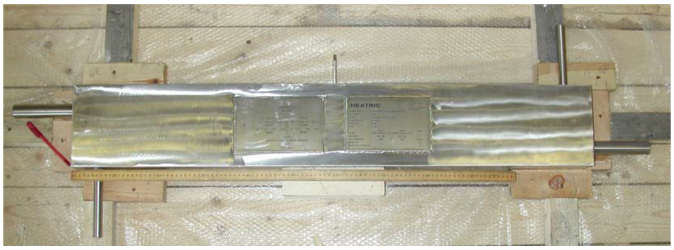


Figure 5: Printed Circuit Heat Exchanger (PCHEX) used by VanMeter [6]

Table 2: PCHEX dimensions estimated by Van meter. [6]. (Items are marked with * the average of values listed by VanMeter was used)

Parameter	Value	Parameter	Value
Number of layers - `N_CR`	21	Number of Columns	53
Channel diameter - `tube`	1.506 mm	Average channel length* - `HX_L`	1304 mm
Plate thickness - `t_1`	1.31 mm	Thickness between channels - `t_2`	0.48 mm
Casing thickness* - `t_casing`	38.55 mm		

5 Examples and Validation

The following examples illustrate the usage of the code and also act to demonstrate the validity

5.1 Micro Channel Heat Exchanger

Printed Circuit Heat Exchangers (PCHEX) are a type of micro-channel heat exchanger. The chemically bonded plates with engraved channels are an efficient way to create effective and compact heat exchangers. However, the compactness can also lead to notable losses due to conduction within the heat exchanger's metallic structure. Furthermore the high thermal gradients cause high mechanical stresses and increase thermal losses.

For the demonstration of the micro-channel heat exchanger model the experimental work by VanMeter [6] was selected. As a first step his work estimates the internal structure of the PCHEX shown in Fig. 5. The data converted to `HX_solver.py` input parameters is summarised in Tab. 2 and the corresponding simulation set-up file is:

```
"""
Example case based on the sCO2 heat exchanger study by:
Josh Van Meter (2006) Experimental Investigation of a Printed Circuit Heat
Exchanger using Supercritical Carbon Dioxide and Water as Heat Transfer Media,
MsC Thesis, Kansas State Univeristy

Used for validation of the micro channel heat exchanger model and for
verification for supercritical fluid simualtions.

Author: Ingo Jahn
Last: Modified 23/03/2017
"""

# set fluid conditions at heat exchanger inlet and outlet
F.fluidH = 'CO2'
F.TH_in = 273.15+88.
F.mdotH = 100./3600
```

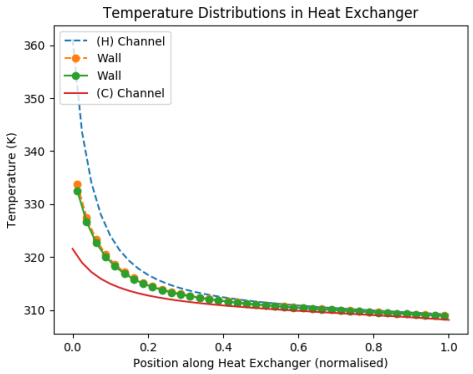
Table 3: Cell number sensitivity study for case with $\dot{m}_{CO_2} = 300\,\mathrm{kg/h}$ and CO_2 outlet temperature at 36 °C.

N_cell	HTC ($\mathrm{W\,m^{-1}\,K^{-1}}$)	% difference	N_cell	HTC ($\mathrm{W\,m^{-1}\,K^{-1}}$)	% difference
5	236.30	28.7	10	186.38	1.5
20	0.31	0	40	183.69	0.07
100	183.57	[n/a]			

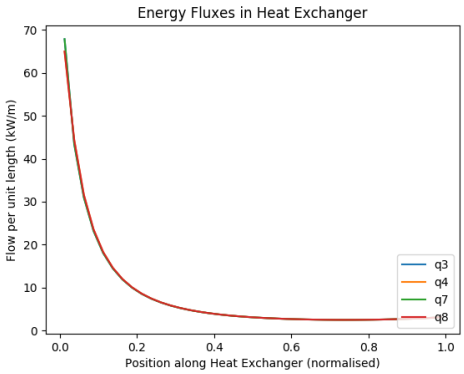
```
F.PH_in = 8.e6
F.fluidC = 'water'
F.TC_in = 273.15+36.0
F.mdotC = 700./3600
F.PC_in = 1.e5
F.T_ext = 295 # External Temperature (K) optional

F.T0 = [ ]

# set geometry for heat exhanger - required settings depend on type
G.HXtype = 'micro-channel'
G.N_R = 21     # number of rows in HX
G.N_C = 53     # number of columns in HX matrix
G.t_1 = 1.31e-3  #
G.t_2 = 0.48e-3 #
G.t_casing = 0.5*(44.6e-3 + 32.5e-3) #
G.HX_L = 0.5* (1.230 + 1.378)     # length of HX (m)
G.d_tube = 1.506e-3 # tube diameter
G.k_wall = 16 # thermal conductivity (W / m /K)
G.epsilonH = 0. # roughness height for H channel
G.epsilonC = 0. # roughness height for C cahannel

# Set modelling parameters
M.N_cell = 40 # number of cells
M.flag_axial = 1
M.external_loss = 0
M.Nu_CorrelationH = 2
M.Nu_CorrelationC = 5
M.f_CorrelationH = 1
M.co_flow = 0
```

To obtain solution independence a sensitivity study on cell number was performed and the results are shown in Tab. 3. This shows that 20 or more cells are required in order to obtain accurate results.

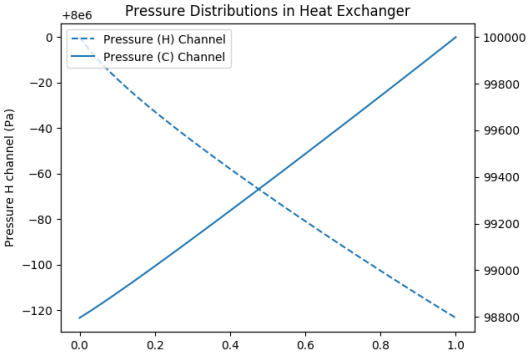
Temperature, pressure and energy flux distributions within the heat exchanger are shown in Fig. 6. These show that significant heat transfer takes place close to the hot CO_2 inlet side, and that heat transfer diminishes closer to the CO_2 outlet. This effect is driven by the nonlinear properties of CO_2 close to the critical point (304.25 K, 7.39 MPa), which also create a pinch point within the heat exchanger. To validate the simulation results the predicted heat transfer coefficient for the current test case: CO_2 inlet conditions 88 °C, 8 MPa, 300 kg/h and CO_2 outlet temperature 36 °C are compared to the corresponding experimental data from Van Meter [6] in Fig. 6d. Two sets of simulation results are shown corresponding to effective area being calculated with Eqns. (4.1a & b). This graphs shows reasonable agreement between the experimental data and predictions. Using the tube surface area (A1-case) consistently under-predicts the heat transfer coefficient. Using the area of the separating surface (A2-case) shows better agreement, however the gradient is somewhat over-predicted. This suggests that the implemented Nusselt number correlations do not fully capture the heat transfer enhancement of the micro channels.



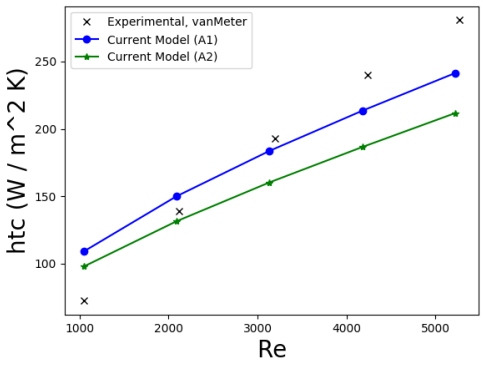
(a) Temperature



(b) Energy Flow



(c) Pressure Distribution



(d) Comparison to heat transfer coefficient data from Van Meter

Figure 6: Solutions for heat exchanger preoperties for case with $\dot{m}_{CO_2} = 300\,\mathrm{kg/h}$ and CO_2 outlet temperature of 36 °C .

Future work should investigate the use of more accurate heat transfer correlations.

5.2 Shell and Tube Heat Exchanger

To illustrate the tool the works by Xie et. al. [7] has been selected. This work studies a counter flow shell and tube heat exchanger, similar to that shown in Fig. 7. To model this type of heat exchanger the key dimensions shown in Tab. 4 are supplied to the `job.py` file. The resulting simulation set-up file is:

```
"""
Example case based on heat exchangers data from.
G.N. Xie, Q.W. Wang, M. Zeng, L.Q. Luo (2007), Heat transfer analysis for
shell-and-tube heat exchangers with experimental data by artificial neural
network approach, Applied Thermal Engineering 27 (2007) 1096-1104

Used for validation of the shell and tube heat exchanger model with oil and water.
The following assumptiosn were made:
- use Therminol T66 as fluid for oil side
- working pressure on both sides is 1 bara (1.e5 Pa)
- to account for forward and backward passage the effective tube length has been
    doubled.
- heat conduction in the casing has been ignored (t_casing = 0)

Author: Ingo Jahn
Last: Modified 23/03/2017
"""

#import CoolProp.CoolProp as CP

# set geometry for heat exhanger - required settings depend on type
G.HXtype = 'shell-tube'
G.N_T = 176/2     # number of tubes (reduced as number given in paper is for both
    directions)
G.d = 8.e-3     # tube inner diameter
G.D = 10.e-3     # tube outer diameter
G.DD = 207.e-3 # shell inner diameter
G.t_casing = 1.e-6 #
G.HX_L = 0.620     # length of HX (m)
G.k_wall = 30 # thermal conductivity (W / m /K)
G.epsilonH = 0. # roughness height for H channel
G.epsilonC = 0. # roughness height for C cahannel

# Boundary Conditions
water = 'water'
T_w = 29.3+273.15    # water inlet temperature
Re_w = 3094          # water inlet Reynolds number
mu_w = CP.PropsSI('V','T',T_w,'P',6.e5,water)     # viscosity
rho_w = CP.PropsSI('D','T',T_w,'P',6.e5,water)    # density
V_w = Re_w * mu_w / G.d / rho_w
mdot_w = rho_w*V_w*G.N_T * G.d**2 / 4*np.pi

oil = 'INCOMP::T66'
T_o = 59.8+273.15    # oil inlet temperature
Re_o = 1825           # oil inlet Reynolds number
mu_o = CP.PropsSI('V','T',T_o,'P',6.e5,oil)     # viscosity
rho_o = CP.PropsSI('D','T',T_o,'P',6.e5,oil)    # density
V_o_max = Re_o * mu_o / G.D / rho_o
#mdot_o = rho_o * V_o* ( G.DD**2/ 4*np.pi - G.N_T * G.D**2 / 4*np.pi )
mdot_o = rho_o * V_o_max* 50e-3**2 * np.pi


# set fluid conditions at heat exchanger inlet and outlet
```

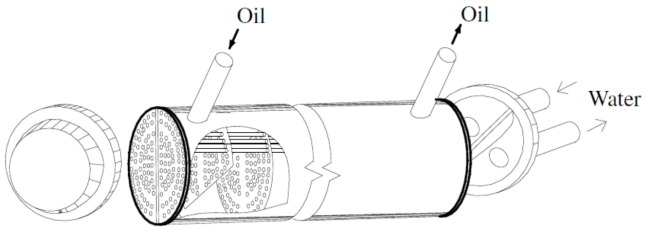


Figure 7: Shell and tube heat exchanger used by Xie et. al. [7]

Table 4: Shell and tube heat exchanger dimensions from Xie et al. [7]. Estimated items are marked with *

Parameter	Value	Parameter	Value
Number of tubes	176	Effective length of tubes	620 mm
Tube outer diameter	10 mm	Tube inner diameter	8 mm
Inner diameter of shell	207 mm		

```
F.fluidH = 'water'
F.TH_in = T_w
F.PH_in = 6.e5
F.PH_out = 6.e5
F.mdotH = mdot_w
F.fluidC = oil
F.TC_in = T_o
F.mdotC = 5.
F.PC_in = 6.e5
F.PC_out = 6.e5
F.mdotC = mdot_o
F.T0 = [ ]

print "Masslfows: mdot_oil=",F.mdotC, '; mdot_water=',F.mdotH, '(kg/s)'

# Set modelling parameters
M.N_cell = 10# number of cells
M.flag_axial = 1
M.external_loss = 0
M.Nu_CorrelationH = 5
M.Nu_CorrelationC = 6
M.f_CorrelationH = 0
M.co_flow = 0
```

The boundary conditions and corresponding results for the 10 cases presented by Xie et al. are shown in Tab. 5. Table 6 shows the results from a cell number (`N_cell`) independency study, which shows that results become grid independent once more than 20 cells are used. The corresponding detailed results for test case (1) are shown in Fig. 8. These plots show the temperature profiles with respect to normalised position along the heat exchanger and also the internal heat fluxes corresponding to Fig. 8b.

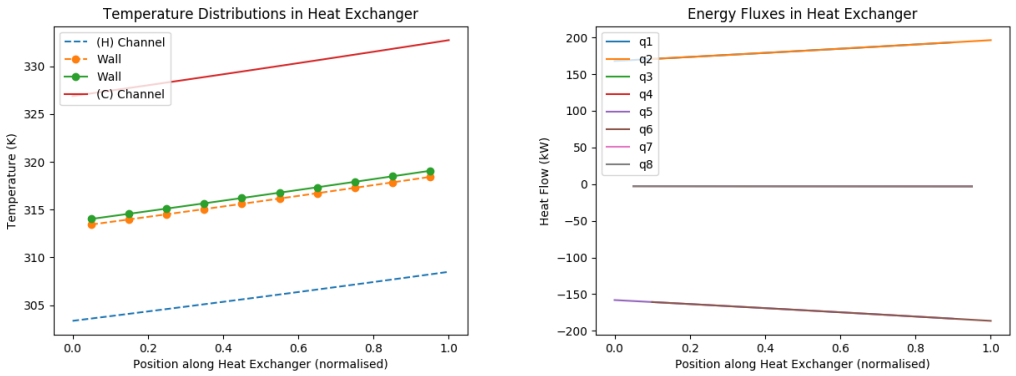
The results are within the range presented by Xie et al. [7], confirming the correct implementation of the model and also it's suitability to analyse heat exchangers. Discrepancies to the actual results are expected to be caused by the use of an incorrect heat transfer oil, as the actual fluid type was not provided by the authors.

Table 5: Tested configurations

Case	$T_{oil,\,out}$ (°C)	Re_{oil}	$T_{water,\,in}$ (°C)	Re_{water}	ΔT_{oil} (K)	ΔT_{water} (K)	HTC W/m/K
(1)	59.6	296	30.2	3010	5.85	5.14	783.04
(2)	58.7	525	28.1	3014	4.16	6.42	964.62
(4)	60.3	697	27.7	2942	3.80	7.43	1031.38
(6)	60.5	821	29.3	3033	3.28	7.52	1087.08
(8)	59.6	1102	30.2	3121	2.43	7.67	1186.70
(13)	59.4	1486	27.9	3022	2.04	8.61	1258.60
(15)	59.8	1825	29.3	3094	1.70	8.75	1317.53

Table 6: Study of results sensitivity to cell number for case (1)

`N_cell`	$T_{oil,\,out}$	$T_{water,\,out}$	Q_{total}
5	329.35 K	320.57 K	−56.16 kW
10	329.36 K	320.56 K	−56.12 kW
20	329.36 K	320.56 K	−96.11 kW



(a) Temperature

(b) Energy Flow

Figure 8: Temperature distributions and energy fluxes within the heat exchanger for case (1).

6 References

References

[1] I. H. Bell, J. Wronski, S. Quoilin, V. Lemort, 2014, *Pure and Pseudo-pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp*, Industrial & Engineering Chemistry Research, 53(6):2498–2508, 2014. doi:10.1021/ie4033999.

[2] S.M. Liao, T.S. Zhao, 2002, *An experimental investigation of convection heat transfer to supercritical carbon dioxide in miniature tubes*, International Journal of Heat and Mass Transfer, 45, (2002), 50255034

[3] S. H. Yoon, J. H. Kim, Y. W. Hwang, M. S. Kim, K. Min, Y. Kim, 2003, *Heat transfer and pressure drop characteristics during the in-tube cooling process of carbon dioxide in the supercritical region*, International Journal of Refrigeration, 26, (2003), 857864

[4] F. O. Incropera, D. P. DeWitt 2007, *Fundamentals of Heat and Mass Transfer*, New York: Wiley, ISBN 978-0-471-45728-2

[5] A. M. Howatson, P. G. Lund, J. D. Todd 1991, *Engineering Tables and data*, London: Chapman & Hall

[6] J. Van Meter, 2006, *Experimental Investigation of a Printed Circuit Heat Exchanger using Supercritical Carbon Dioxide and Water as the Heat Transfer Media*, Master of Science Thesis, Kansas State University, USA

[7] G. N. Xie, Q. W. Wang, M. Zeng, L. Q. Luo, 2007, *Heat transfer analysis for shell-and-tube heat exchangers with experimental data by artificial neural network approach*, Applied Thermal Engineering, 27, (2007), 1096-1104

7 Appendix

7.1 Example Input file job_example.py

```
"""
Template input file for HX_solver.py
"""

# set fluid conditions at heat exchanger inlet and outlet
F.fluidH = 'CO2'
F.TH_in = 756.170449065
F.mdotH = 5.
F.PH_in = 7.68e6
F.PH_out = 7.68e6
F.TC_in = 324.91481002
F.mdotC = 5.
F.PC_in = 12.e6
F.PC_out = 12.e6
F.T_ext = 295 # External Temperature (K) optional

F.T0 = [ ]

```

```
19 # set geometry for heat exhanger - required settings depend on type
20 G.HXtype = 'micro-channel'
21 G.N_R = 100     # number of rows in HX
22 G.N_C = 400     # number of columns in HX matrix
23 G.t_1 = 2e-3  #
24 G.t_2 = 0.5e-3 #
25 G.t_casing = 5e-3 #
26 G.HX_L = 1.     # length of HX (m)
27 G.d_tube = 1.5e-3 # tube diameter
28 G.k_wall = 16 # thermal conductivity (W / m /K)
29 G.epsilonH = 0. # roughness height for H channel
30 G.epsilonC = 0. # roughness height for C cahannel
31
32 # Set modelling parameters
33 M.N_cell = 10 # number of cells
34 M.flag_axial = 1
35 M.external_loss = 0
36 M.Nu_CorrelationH = 2
37 M.Nu_CorrelationC = 2
38 M.f_CorrelationH = 1
39 M.co_flow = 0
```

7.2 Source Code HX_solver.py

```
1 #! /usr/bin/env python
2 """
3 Python Code to evaluate on- and off-design performance of heat exchangers.
4
5 Function has two oeprating modes:
6 (1) Stand alone
7 This evaluates the heat exchanger performance and can be used to plot
8 temperature traces inside the heat exchanger.
9
10 (2) imported
11 The function is can be called by Cycle.py to allow the quasi-steady evaluation
12 of heat exchanger performance as part of Cycle off-design modelling.
13
14 Version: 1.0
15 Author: Ingo Jahn
16 Last Modified: 26/03/2017
17 """
18
19 import numpy as np
20 import CoolProp.CoolProp as CP
21 import scipy as sci
22 from scipy import optimize
23 import matplotlib.pyplot as plt
24 import sys as sys
25 import os as os
26 from getopt import getopt
27
28 ###
29 ###
30 class Fluid:
31     def __init__(self):
32         self.fluidH  = []
33         self.fluidC  = []
34         self.TH_in  = [] # (K)
```

```
35          self.mdotH  = [] # (kg/s)
36          self.PH_in  = [] # pressure (Pa)
37          self.PH_out = [] # pressure (Pa)
38          self.TC_in  = [] # (K)
39          self.mdotC  = [] # (kg/s)
40          self.PC_in  = [] # pressure (Pa)
41          self.PC_out = [] # pressure (Pa)
42          self.T_ext = [] # ambient surroundign temperature
43          self.T0 = []
44      ###
45      def check(self,M):
46          if not self.fluidH and not self.fluidC:
47              raise MyError('Neither F.fluidH or F.fluidC specified')
48          if self.fluidH and not self.fluidC:
49              self.fluidC = self.fluidH
50              print 'fluidC not specified and set to equal fluidH'
51          if self.fluidC and not self.fluidH:
52              self.fluidH = self.fluidC
53              print 'fluidH not specified and set to equal fluidC'
54          if not self.TH_in:
55              raise MyError('F.TH_in not specified correctly')
56          if not self.mdotH:
57              raise MyError('F.mdot_H not specified')
58          if not self.PH_in:
59              raise MyError('F.PH_in not specified')
60          if not self.PH_out:
61              self.PH_out = self.PH_in
62              print 'PH_out not specified and set to equal PH_in'
63          if not self.TC_in:
64              raise MyError('F.TC_in not specified')
65          if not self.mdotC:
66              self.mdotC = self.mdotH
67              print 'mdotC not specified and set to equal mdotH'
68          if self.mdotC < 0. or self.mdotH < 0.:
69              raise MyError('Mass flow rates in both channels needs to be > 0.')
70          if not self.PC_in:
71              raise MyError('F.PC_in not specified')
72          if not self.PC_out:
73              self.PC_out = self.PC_in
74              print 'PC_out not specified and set to equal PC_in'
75          #if M.external_loss is 1:
76          #    if not self.T_ext:
77          #        raise MyError('T_ext not specified')
78          #
79          if M.print_flag:
80              print 'Check of Fluid input parameters: Passed'
81
82      ###
83      def get_T0(self,M):
84          """
85          function to set initial/old conditions
86          """
87          if len(self.T0) is 0:
88              T0 = np.zeros((4*M.N_cell))
89
90
91              # set TWH
92              T0[M.N_cell:2*M.N_cell] = np.ones(M.N_cell) * 0.5*(self.TH_in+self.
                    TC_in)
93              # set TWC
94              T0[2*M.N_cell:3*M.N_cell] = np.ones(M.N_cell) * 0.5*(self.TH_in+self.
                    TC_in)
```

```
95              if M.co_flow:
96                  # set TH
97                  T0[0:M.N_cell] = self.TH_in + np.arange(M.N_cell)/float(M.N_cell)
                        *(self.TC_in-self.TH_in)*0.25
98                  # set TC
99                  T0[3*M.N_cell:4*M.N_cell] = self.TC_in - np.arange(M.N_cell)/float
                        (M.N_cell)*(self.TC_in-self.TH_in)*0.25
100             else:
101                 # set TH
102                 T0[0:M.N_cell] = self.TH_in + np.arange(M.N_cell)/float(M.N_cell)
                        *(self.TC_in-self.TH_in)*0.5
103                 # set TC
104                 T0[3*M.N_cell:4*M.N_cell] = self.TC_in - (1.-np.arange(M.N_cell)/
                        float(M.N_cell))*(self.TC_in-self.TH_in)*0.5
105                 #T0[4*M.N_cell:5*M.N_cell] = np.ones(M.N_cell) * 0.5* (self.PH_in
                        + self.PH_out)
106             #T0[5*M.N_cell:6*M.N_cell] = np.ones(M.N_cell) * 0.5* (self.PC_in +
                    self.PC_out)
107             return T0
108         else:
109             return self.T0
110 ###
111 ###
112 class Model:
113     def __init__(self):
114         self.optim ='root:hybr'
115         self.N_cell = [] # number of cells
116         self.co_flow = 0  # default is to analyse counterflow heat exchangers. Set
                to 1 for co flow
117         self.flag_axial = [] # whether axial heat conduction is considered
118         self.external_loss = [] # whether external heat loss is considered
119         self.Nu_CorrelationH = [] # set correlation for heat transfer in H channel
120         self.Nu_CorrelationC = [] # set correlation for heat transfer in C channel
121         self.f_CorrelationH = [] # set correlation for friction factor in H
                channel
122         self.f_CorrelationC = [] # set correlation for friction factor in C
                channel
123         self.H_dP_poly = [] # polynominal coefficients for pressure drop in H
                channel
124         self.C_dP_poly = [] # polynominal coefficients for pressure drop in H
                channel
125     ###
126     def check(self):
127         if not self.N_cell:
128             raise MyError('M.N_cell not specified')
129         if not self.flag_axial:
130             self.flag_axial = 0
131             print 'M.flag_axial no specified defaulting to 0'
132         if not self.Nu_CorrelationH:
133             raise MyError('M.Nu_CorrelationH not specified')
134         if not self.Nu_CorrelationC:
135             self.Nu_CorrelationC = self.Nu_CorrelationH
136             print 'Nu_CorrelationC not specified and set to equal Nu_CorrelationH'
137         if not isinstance(self.f_CorrelationH,int):
138             raise MyError('M.f_CorrelationH not specified')
139         if not isinstance(self.f_CorrelationC,int):
140             self.f_CorrelationC = self.f_CorrelationH
141             print 'f_CorrelationC not specified and set to equal f_CorrelationH'
142         if not (self.co_flow == 0 or self.co_flow == 1):
143             raise MyError('M.co_flow not defined correctly. Set to 0 or 1')
144         if len(self.H_dP_poly) > 0 and self.f_CorrelationH != 0:
```

```
145             raise MyError('Pressure drop polynominal H_dP_poly can only be used in
                    conjunction with f_CorrelationH = 0')
146         if len(self.C_dP_poly) > 0 and self.f_CorrelationC != 0:
147             raise MyError('Pressure drop polynominal C_dP_poly can only be used in
                    conjunction with f_CorrelationC = 0')
148         #
149         if self.print_flag:
150             print 'Check of Model input parameters: Passed'
151 ###
152 ###
153     def set_poly(self,F):
154         # use polynominal to update outlet pressure
155         if len(self.H_dP_poly) > 0:
156             temp = []
157             for a in range(len(self.H_dP_poly)):
158                 temp.append(self.H_dP_poly*F.mdotC**a)
159             F.PC_out = sum(temp)
160 
161         if len(self.C_dP_poly) > 0: # if polynominal is specified use this to calc
                PH_out
162             temp = []
163             for a in range(len(self.C_dP_poly)):
164                 temp.append(self.C_dP_poly*F.mdotC**a)
165             F.PC_out = sum(temp)
166 ###
167 ###
168 class Geometry:
169     def __init__(self):
170         self.HXtype = []
171         self.k_wall = [] # thermal conductivity (W / mk)
172         self.type_list = ['micro-channel','shell-tube']
173     ###
174     def check_initialise(self,M):
175         if not self.HXtype:
176             raise MyError('G.HXtype not specified')
177         if not any(self.HXtype in s for s in self.type_list):
178             raise MyError('Specified type in G.HXtype not supported. Use:'+self.
                    type_list)
179 
180         # look at microchannel case
181         if self.HXtype == 'micro-channel':
182             self.micro_check()
183             self.micro_init()
184         #
185         if self.HXtype == 'shell-tube':
186             self.shell_tube_check()
187             self.shell_tube_init()
188         #
189         if M.print_flag:
190             print 'Check of Geometry input parameterrs: Passed'
191     ###
192     def micro_check(self):
193         if not self.N_R:
194             raise MyError('G.N_R not specified')
195         if not self.N_C:
196             raise MyError('G.N_C not specified')
197         if not self.t_1:
198             raise MyError('G.t_1 not specified')
199         if not self.t_2:
200             raise MyError('G.t_2 not specified')
201         if not self.t_casing:
202             raise MyError('G.t_casing not specified')
```

```
203         if not self.HX_L:
204             raise MyError('G.HX_L not specified')
205         if not self.d_tube:
206             raise MyError('G.d_tube not specified')
207         if not self.k_wall:
208             raise MyError('G.k_wall not specified')
209     ###
210     def micro_init(self):
211         self.Area = self.N_C * (self.d_tube + self.t_2) * (2*self.N_R-1) * self.
                HX_L
212         #self.Area = self.N_C * self.N_R * (self.d_tube * np.pi) * self.HX_L
213 
214         self.AH = self.N_R*self.N_C * self.d_tube**2/4 *np.pi # total flow area (
                m2)
215         self.Lc_H = self.d_tube # characteristic length (m)
216         self.Dh_H = self.d_tube
217         self.AC = self.N_R*self.N_C * self.d_tube**2/4 *np.pi # total flow area (
                m2)
218         self.Lc_C = self.d_tube # characteristic length (m)
219         self.Dh_C = self.d_tube
220 
221         self.t_wall = self.t_1 # (m)
222 
223         L1 = self.N_C*self.d_tube  + (self.N_C-1)*self.t_2    + 2*self.t_casing
224         L2 = 2*self.N_R*self.d_tube+ (2*self.N_R-1)*self.t_1 + 2*self.t_casing
225         self.A_wall = L1*L2 - self.AH - self.AC
226 
227         self.L_wall = self.HX_L
228     ###
229     def shell_tube_check(self):
230         if not self.N_T:
231             raise MyError('G.N_T not specified')
232         if not self.d:
233             raise MyError('G.d not specified')
234         if not self.D:
235             raise MyError('G.D not specified')
236         if not self.DD:
237             raise MyError('G.DD not specified')
238         if not self.t_casing:
239             raise MyError('G.t_casing not specified')
240         if not self.HX_L:
241             raise MyError('G.HX_L not specified')
242         if not self.k_wall:
243             raise MyError('G.k_wall not specified')
244     ###
245     def shell_tube_init(self):
246         self.Area = 0.5*(self.d +self.D) * np.pi * self.HX_L * self.N_T
247         self.AH = self.N_T * self.d**2/4 *np.pi # total flow area (m2)
248         self.Lc_H = self.d # characteristic length (m)
249         self.Dh_H = self.d # hydraulic diameter
250         self.AC = self.DD**2/4*np.pi - self.N_T* self.D**2/4.*np.pi # total flow
                area (m2)
251         self.AC = self.AC
252         self.Lc_C = self.D # characteristic length (m)
253         Perimeter = self.DD * np.pi + self.N_T * self.D*np.pi
254         self.Dh_C = 4.* self.AC / Perimeter # hydraulic diameter
255 
256         self.t_wall = (self.D-self.d)/2. # (m)
257 
258         self.A_wall = self.DD*np.pi*self.t_wall + self.N_T * (self.D**2 - self.d
                **2) / 4.*np.pi
259 
```

```
260         self.L_wall = self.HX_L
261 ##
262 ##
263 def calc_Nu(P,Tm,Tp,Tw, mdot,A_total,L_c,fluid ,Correlation):
264     """
265     function to calculate local Nusselt number based on bulk flow properties
266     Inputs:
267     P - bulk pressure (Pa)
268     Tm - bulk temperature to left (K)
269     Tp - bulk temperature to right (K)
270     Tw - temperature of wall (K)
271     mdot - total mass flow rate (kg/s)
272     A_total - total flow area (m**2)
273     L_c - characteristic length (m)
274     fluid - fluid type
275     Correlation - select correlation to be used:
276         1 - Yoon correlation for sCO2 in pipes for Tb > Tpc
277         2 - S.M. Liao and T.S Zhaou correlation for micorchannels
278             http://www.me.ust.hk/~mezhao/pdf/33.PDF
279             - horizontal pipes
280         3 - S.M. Liao and T.S Zhaou correlation for micorchannels
281             http://www.me.ust.hk/~mezhao/pdf/33.PDF
282             - vertical pipes, upwards flow
283         4 - - S.M. Liao and T.S Zhaou correlation for micorchannels
284             http://www.me.ust.hk/~mezhao/pdf/33.PDF
285             - vertical pipes, downwards flow
286
287     Outputs:
288     Nu - Nussel Number
289     """
290     Tb = 0.5*(Tm+Tp) # bulk temperature
291
292     # calculate Prandl number
293     Pr = CP.PropsSI('PRANDTL','P',P,'T', Tb ,fluid)
294
295     # claculate Reynolds number
296     rho_b = CP.PropsSI('DMASS','P',P,'T', Tb ,fluid)
297     U = abs( mdot / (rho_b * A_total))
298     mu_b = CP.PropsSI('VISCOSITY','P',P,'T', Tb ,fluid)
299     Re = rho_b * U * L_c / mu_b
300
301     if Correlation is 1:
302         # Yoon correlation for horizontal pipes
303         Nu = 0.14 * Re**0.69 * Pr**0.66
304     elif Correlation is 2:
305         # Liao correlation for horizontal pipes
306         rho_w = CP.PropsSI('DMASS','P',P,'T', Tw ,fluid)
307         Gr = abs( 9.80665 * (rho_b-rho_w)*rho_b*L_c**3 / mu_b**2)
308         Cp_b = CP.PropsSI('CPMASS','P',P,'T', Tb ,fluid)
309         i_b = CP.PropsSI('UMASS','P',P,'T', Tb ,fluid)
310         i_w = CP.PropsSI('UMASS','P',P,'T', Tw ,fluid)
311         if Tw == Tb:
312             Nu = 0.
313         else:
314             Cp_bar = (i_w-i_b) / (Tw-Tb)
315             #print 'Gr: ', Gr, 'Cp_bar: ', Cp_bar
316             Nu = 0.124 * Re**0.8 * Pr**0.4 * (Gr/Re**2)**0.203 * (rho_w/rho_b)
                    **0.842 * (Cp_bar / Cp_b)**0.384
317     elif Correlation is 3:
318         # Liao correlation for vertical pipes - upwards flow
319         rho_w = CP.PropsSI('DMASS','P',P,'T', Tw ,fluid)
320         rho_mid = CP.PropsSI('DMASS','P',P,'T', 0.5*(Tw+Tb) ,fluid)
```

```
321         rho_m = 1/(Tw-Tb) * (Tw-Tb)/6. * (rho_b + 4*rho_mid + rho_w)  #
                integration using Simpsons rule
322         Gr_m = abs( 9.80665 * (rho_b-rho_m)*rho_b*L_c**3 / mu_b**2)
323         Cp_b = CP.PropsSI('CPMASS','P',P,'T', Tb ,fluid)
324         i_b = CP.PropsSI('UMASS','P',P,'T', Tb ,fluid)
325         i_w = CP.PropsSI('UMASS','P',P,'T', Tw ,fluid)
326         if Tw == Tb:
327             Nu = 0.
328         else:
329             Cp_bar = (i_w-i_b) / (Tw-Tb)
330             #print 'Gr: ', Gr, 'Cp_bar: ', Cp_bar
331             Nu = 0.354 * Re**0.8 * Pr**0.4 * (Gr_m/Re**2.7)**0.157 * (rho_w/rho_b)
                    **1.297 * (Cp_bar / Cp_b)**0.296
332     elif Correlation is 4:
333         # Liao correlation for vertical pipes - downwards flow
334         rho_w = CP.PropsSI('DMASS','P',P,'T', Tw ,fluid)
335         rho_mid = CP.PropsSI('DMASS','P',P,'T', 0.5*(Tw+Tb) ,fluid)
336         rho_m = 1/(Tw-Tb) * (Tw-Tb)/6. * (rho_b + 4*rho_mid + rho_w)  #
                integration using Simpsons rule
337         Gr_m = abs( 9.80665 * (rho_b-rho_m)*rho_b*L_c**3 / mu_b**2)
338         Cp_b = CP.PropsSI('CPMASS','P',P,'T', Tb ,fluid)
339         i_b = CP.PropsSI('UMASS','P',P,'T', Tb ,fluid)
340         i_w = CP.PropsSI('UMASS','P',P,'T', Tw ,fluid)
341         if Tw == Tb:
342             Nu = 0.
343         else:
344             Cp_bar = (i_w-i_b) / (Tw-Tb)
345             #print 'Gr: ', Gr, 'Cp_bar: ', Cp_bar
346             Nu = 0.643 * Re**0.8 * Pr**0.4 * (Gr_m/Re**2.7)**0.186 * (rho_w/rho_b)
                    **2.154 * (Cp_bar / Cp_b)**0.751
347     elif Correlation is 5:
348         # For flow in circular pipes (from HLT) use for incompressible fluids (
                water/oil)
349         # Dittus Boelter Equation
350         if Tw > Tb: # heating of fluid
351             n = 0.3
352         else:       # cooling of fluid
353             n = 0.4
354         Nu = 0.023 * Re**0.8 * Pr**n
355     elif Correlation is 6:
356         # Correlation for shaell side as per paper by Xie et al.
357         C = 0.16442; m=0.65582
358         e = np.exp( C + m * np.log(Re) )
359         Nu = e * Pr**(1./3.)
360     elif Correlation is 7:
361         if Re == 0. or Re < 0.0001:
362             # use Natural Convection relationship
363             rho_w = CP.PropsSI('DMASS','P',P,'T', Tw ,fluid)
364             beta = 1./Tb
365             Gr = abs( 9.80665 * rho_w**2 * L_c**3 * (Tw - Tb) * beta / mu_b**2)
366             GrPr = Gr*Pr
367             if GrPr <= 1e9:
368                 C = 0.53; n=1/4
369             else:
370                 C = 0.126; n=1/3
371             Nu = C * GrPr **n
372         if Re > 0.0001 and Re <= 0.004:
373              C = 0.437; n = 0.0985
374         elif Re >0.004 and Re <= 0.09:
375              C = 0.565; n = 0.136
376         elif Re > 0.09 and Re <= 1.:
377             C = 0.8; n = 0.280
```

```
378         elif Re > 1.0 and Re <= 35.:
379             C = 0.795; n= 0.384
380         elif Re > 35. and Re <= 5000.:
381             C = 0.583; n = 0.471
382         elif Re > 5000. and Re <= 50000.:
383             C = 0.148; n = 0.633
384         elif Re > 50000. and Re <= 200000:
385             C = 0.0208; n = 0.814
386         else:
387             raise MyError('Correlation outside range of valid Nusselt numbers.')
388 
389     else:
390         raise MyError('Correlation option for Nusselt number calculation not
                implemented.')
391 
392     return Nu
393 ##
394 ##
395 def calc_friction(P, Tm, Tp, mdot, A_total, Dh, fluid, Correlation, epsilon = 0.):
396     """
397     function to calculate local friction factor based on local geometry and bulk
            flow properties
398     Inputs:
399     P - bulk pressure (Pa)
400     Tm - bulk temperature to left (K)
401     Tp - bulk temperature to right (K)
402     mdot - total mass flow rate (kg/s)
403     A_total - total flow area (m**2)
404     Dh - Hydraulic Diameter (m)
405     fluid - fluid type
406     Correlation - select correlation to be used:
407         1 - autmatically switches between laminar and turbulent flow
408         2 - laminar flow - circular pipe
409         3 - turbulent flow - rough pipes (Haaland's formula)
410     epsilon - roughness height (m)
411 
412     Outputs:
413     f - friction factor
414     """
415     Tb = 0.5*(Tm+Tp) # bulk temperature
416 
417     # claculate Reynolds number
418     rho_b = CP.PropsSI('DMASS','P',P,'T', Tb ,fluid)
419     U = abs(mdot / (rho_b * A_total))
420     mu_b = CP.PropsSI('VISCOSITY','P',P,'T', Tb ,fluid)
421     Re = rho_b * U * Dh / mu_b
422     if Correlation is 1:
423         if Re < 2300:
424             f = 64. / Re
425         else:
426             temp = np.log10( (epsilon/Dh / 3.7)*1.11 + (6.9/Re ))
427             f = (-1.8 * temp )**-2.
428     elif Correlation is 2:
429         # laminar pipe flow
430         f = 64. / Re
431     elif Correlation is 3:
432         # tubulent rough pipes
433         temp = np.log10( (epsilon/Dh / 3.7)*1.11 + (6.9/Re ))
434         f = (-1.8 * temp )**-2.
435 
436     else:
```

```
437         raise MyError('Correlation option for friction factor calculation not
                implemented.')
438
439     # print Gr / Re**2
440
441     return f
442 ###
443 ###
444 def equations(T, M, G, F, flag=0):
445     """
446     function that evaluates the steady state energy balance for each cell
447     Inputs:
448     T - vector containg temperatures and pressures at various interface points
449     M - class containign model parameters
450     G - class containing geometry parameters
451     F - class containing fluid boundary conditions
452     flag - allows operation fo function to be altered
453             0 - default for operation
454             1 - output temperature and heat fluxes
455             2 - returns pressure traces
456     Outputs:
457     error - vector containing miss-balance in energy equations for different
            lcoations
458     """
459     TH, TWH, TWC, TC= open_T(T,F.TH_in,F.TC_in,F.PH_in,F.PC_in,M,F)
460
461     # print 'Temperature',TH,TWH,TC
462     if flag is 1:
463         Q1 = []; Q2 = []; Q3 = []
464         Q4 = []; Q5 = []; Q6 = []
465         Q7 = []; Q8 = []
466
467     error = np.zeros(4*M.N_cell)
468
469     # calculate Pressure distribution in both pipes based on current temperatures
470     # pressure is calculated in flow direction.
471     PH = [F.PH_in]; PC=[F.PC_in]
472     for i in range(M.N_cell):
473         if M.f_CorrelationH == 0: # apply linear pressure drop if no correlation
                is specificed
474             PH.append(F.PH_in -  (F.PH_in - F.PH_out) / (M.N_cell-1.) * (i+1))
475         elif M.f_CorrelationH == 1:
476             # calculate pressure drop due to friction
477             f= calc_friction(PH[i-1],TH[i],TH[i+1],F.mdotH,G.AH,G.Dh_H, F.fluidH ,
                    M.f_CorrelationH , G.epsilonH)
478             rhoH = CP.PropsSI('D','P', PH[i-1], 'T', TH[i] ,F.fluidH)
479             V = F.mdotH / rhoH / G.AH # calculate flow velocity
480             h_f = f * G.HX_L/(M.N_cell+1.) /G.Dh_H * V*V / (2.*9.81) # calculate
                    friction head loss
481             dP = h_f * rhoH * 9.81
482             PH.append(PH[i] - dP)
483         else:
484             raise MyError('Corelation type not implemented')
485
486         if M.f_CorrelationC == 0: # apply linear pressure drop if no correlation
                is specificed
487             PC.append(F.PC_in - (F.PC_in - F.PC_out) / (M.N_cell-1.) * (i+1))
488         elif M.f_CorrelationC == 1:
489             if M.co_flow:
490                 f= calc_friction(PC[i-1],TC[i],TC[i+1],F.mdotC,G.AC,G.Dh_C, F.
                        fluidC , M.f_CorrelationC , G.epsilonC)
491                 rhoC = CP.PropsSI('D','P', PC[i-1], 'T', TC[i] ,F.fluidC)
```

```
492                 V = F.mdotC / rhoC / G.AC # calculate flow velocity
493                 h_f = f * G.HX_L/(M.N_cell+1.) /G.Dh_C * V*V / (2.*9.81) #
                        calculate friction head loss
494                 dP = h_f * rhoC * 9.81
495                 PC.append(PC[i] - dP)
496             else:
497                 # calculate pressure drop due to friction
498                 j = M.N_cell - i - 1  #
499                 f= calc_friction(PC[i-1],TC[j],TC[j+1],F.mdotC,G.AC,G.Dh_C, F.
                        fluidC , M.f_CorrelationC , G.epsilonC)
500                 rhoC = CP.PropsSI('D','P', PC[i-1], 'T', TC[j] ,F.fluidC)
501                 V = F.mdotC / rhoC / G.AC # calculate flow velocity
502                 h_f = f * G.HX_L/(M.N_cell+1.) /G.Dh_C * V*V / (2.*9.81) #
                        calculate friction head loss
503                 dP = h_f * rhoC * 9.81
504                 PC.append(PC[i] - dP)
505         else:
506             raise MyError('Corelation type not implemented')
507
508
509     if not M.co_flow: #
510         # reverse direction of PC as flow is from i = -1 to i = 0
511         PC = list(reversed(PC))
512
513     #print 'PH', PH
514     #print 'PC', PC
515
516     # calculate energy balance for high pressure stream (H); low pressure stream (
            C) and dividing wall
517     for i in range(M.N_cell):
518
519         #print 'In HX_solver'
520         #print 'TH', TH
521         #print 'PH', PH
522         kH = CP.PropsSI('CONDUCTIVITY','P', (0.5*(PH[i]+PH[i+1])), 'T', (0.5*(TH[i
                ]+TH[i+1])) ,F.fluidH)
523         kHm = CP.PropsSI('CONDUCTIVITY','P', PH[i],                 'T', TH[i] ,F.
                fluidH)
524         kHp = CP.PropsSI('CONDUCTIVITY','P', PH[+1],                'T', TH[i+1] ,F
                .fluidH)
525         kC = CP.PropsSI('CONDUCTIVITY','P', (0.5*(PC[i]+PC[i+1])), 'T', (0.5*(TC[i
                ]+TC[i+1])) ,F.fluidC)
526         kCm = CP.PropsSI('CONDUCTIVITY','P', PC[i],                 'T', TC[i] ,F.
                fluidC)
527         kCp = CP.PropsSI('CONDUCTIVITY','P', PC[i+1],               'T', TC[i+1] ,F
                .fluidC)
528
529         NuH = calc_Nu((0.5*(PH[i]+PH[i+1])),TH[i],TH[i+1],TWH[i], F.mdotH, G.AH, G
                .Lc_H, F.fluidH, M.Nu_CorrelationH)
530         NuC = calc_Nu((0.5*(PC[i]+PC[i+1])),TC[i],TC[i+1],TWC[i], F.mdotC, G.AC, G
                .Lc_C, F.fluidC, M.Nu_CorrelationC)
531
532         hH = NuH * kH / G.Lc_H
533         hC = NuC * kC / G.Lc_C
534
535         # heat trasnfer in H channel
536         q1_conv = CP.PropsSI('HMASS','P',PH[i],'T',TH[i],F.fluidH) * F.mdotH
537         if i == 0:
538             q1_cond = - kHm * G.AH/ (G.L_wall/M.N_cell/2.) * ( 0.5*(TH[i] + TH[i
                    +1]) - TH[i] )
539         else:
```

```
540             q1_cond = - kHm * G.AH/ (G.L_wall/M.N_cell)     * ( 0.5*(TH[i] + TH[i
                    +1]) - 0.5*(TH[i] + TH[i-1]) )
541         q1 = q1_conv + q1_cond
542
543         q2_conv = CP.PropsSI('HMASS','P',PH[i+1],'T',TH[i+1],F.fluidH) * F.mdotH
544         if i == M.N_cell-1:
545             q2_cond = - kHp * G.AH/ (G.L_wall/M.N_cell/2.) * (       TH[i+1]
                            - 0.5*(TH[i] + TH[i+1]) )
546         else:
547             q2_cond = - kHp * G.AH/ (G.L_wall/M.N_cell)     * ( 0.5*(TH[i+1] + TH[i
                    +2]) - 0.5*(TH[i] + TH[i+1]) )
548         q2 = q2_conv + q2_cond
549         q3 = hH * G.Area/M.N_cell * (0.5*(TH[i]+TH[i+1]) - TWH[i])
550
551         # Heat Transfer in wall
552         q7_temp = (G.k_wall * G.Area/M.N_cell)/G.t_wall * (TWH[i] - TWC[i])
553         if M.flag_axial is 1:
554             if i == 0:
555                 q7_p = - G.k_wall* G.A_wall/ (G.L_wall/M.N_cell) * ( 0.5*(TWH[i
                            +1]+TWC[i+1]) - 0.5*(TWH[i]  +TWC[i]  ))
556                 q7_m = 0.
557             elif i == M.N_cell-1:
558                 q7_p = 0.
559                 q7_m = - G.k_wall* G.A_wall/ (G.L_wall/M.N_cell) * ( 0.5*(TWH[i]
                            +TWC[i]  ) - 0.5*(TWH[i-1]+TWC[i-1]))
560             else:
561                 q7_p = - G.k_wall* G.A_wall/ (G.L_wall/M.N_cell) * ( 0.5*(TWH[i
                            +1]+TWC[i+1]) - 0.5*(TWH[i]  +TWC[i]  ))
562                 q7_m = - G.k_wall* G.A_wall/ (G.L_wall/M.N_cell) * ( 0.5*(TWH[i]
                            +TWC[i]  ) - 0.5*(TWH[i-1]+TWC[i-1]))
563         else:
564             q7_p = 0.
565             q7_m = 0.
566         q7h = q7_temp - 0.5 * (-q7_p +q7_m)
567         q7c = q7_temp + 0.5 * (-q7_p +q7_m)
568
569         q4 = hC * G.Area/M.N_cell * (TWC[i] - 0.5*(TC[i]+TC[i+1]))
570         qC_cond = - kC * G.AC/ (G.L_wall/M.N_cell) * ( TC[i+1] - TC[i])
571
572         # Heat Transfer in C Channel
573         q5_conv = CP.PropsSI('HMASS','P',PC[i],'T',TC[i],F.fluidC) * -F.mdotC
574         if i == 0:
575             q5_cond = - kCm * G.AC/ (G.L_wall/M.N_cell/2.) * ( 0.5*(TC[i] + TC[i
                    +1]) - TC[i] )
576         else:
577             q5_cond = - kCm * G.AC/ (G.L_wall/M.N_cell)     * ( 0.5*(TC[i] + TC[i
                    +1]) - 0.5*(TC[i] + TC[i-1]) )
578         q5 = q5_conv + q5_cond
579         q6_conv = CP.PropsSI('HMASS','P',PC[i+1],'T',TC[i+1],F.fluidC) * -F.mdotC
580         if i == M.N_cell-1:
581             q6_cond = - kCp * G.AC/ (G.L_wall/M.N_cell/2.) * (       TC[i+1]
                            - 0.5*(TC[i] + TC[i+1]) )
582         else:
583             q6_cond = - kCp * G.AC/ (G.L_wall/M.N_cell)     * ( 0.5*(TC[i+1] + TC[i
                    +2]) - 0.5*(TC[i] + TC[i+1]) )
584         q6 = q6_conv + q6_cond
585
586         # calculate mis-match in energy fluxes
587         #print q1, q2, q3
588         error[i] = q1-q2-q3
589         error[M.N_cell+i] = q3-q7h
590         error[2*M.N_cell+i] = q4-q7c
```

```
591         error[3*M.N_cell+i] = q4+q5-q6
592
593         if flag is 1:
594             Q1.append(q1); Q2.append(q2); Q3.append(q3)
595             Q4.append(q4); Q5.append(q5); Q6.append(q6)
596             Q7.append(q7h); Q8.append(q7c)
597
598         #print q1,q2,q3,q4,q5,q6,q7
599         #print 'Error', error
600         #print i
601
602     if flag is 0:
603         return error
604     elif flag is 1:
605         return error, T, Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8
606     elif flag is 2:
607         return PH, PC
608     else:
609         raise MyError('flag option not defined.')
610 ###
611 ###
612 def open_T(T,TH_in,TC_in,PH_in,PC_in,M,F):
613     """
614     function to unpack the Temperature vector T into the 6 vectors
615     TH, TWH, TWC, TC, PH, PC
616     """
617     N_cell = M.N_cell
618
619     TH = np.zeros(N_cell+1)
620     TWH = np.zeros(N_cell)
621     TWC = np.zeros(N_cell)
622     TC = np.zeros(N_cell+1)
623     TH[0] = TH_in
624     TH[1:N_cell+2] = T[0:N_cell]
625     TWH = T[N_cell:2*N_cell]
626     TWC = T[2*N_cell:3*N_cell]
627     if M.co_flow:
628         TC[0] = TC_in
629         TC[1:N_cell+2] = T[3*N_cell:4*N_cell]
630     else:
631         TC[0:N_cell] = T[3*N_cell:4*N_cell]
632         TC[N_cell] =TC_in
633
634
635
636     return TH,TWH,TWC,TC #,PH,PC
637 ###
638 ###
639 def main(uoDict):
640     """
641     main function
642     """
643     # create string to collect warning messages
644     warn_str = "\n"
645
646     # main file to be executed
647     jobFileName = uoDict.get("--job", "test")
648
649     # strip .py extension form jobName
650     jobName = jobFileName.split('.')
651     jobName = jobName[0]
652
```

```
653     # create classes to store input data
654     M = Model()
655     F = Fluid()
656     G = Geometry()
657
658     # set print_flag (can be overwritten from jobfile)
659     M.print_flag = 1
660     if uoDict.has_key("--noprint"):
661         M.print_flag = 0
662
663     # Execute jobFile, this creates all the variables
664     execfile(jobFileName,globals(),locals())
665
666     if M.print_flag:
667         print "Input data file read"
668
669     # Check that required input data has been provided
670     M.check()
671     F.check(M)
672     M.set_poly(F)
673     G.check_initialise(M)
674
675     # Initialise temperature vector
676     T0 = F.get_T0(M)
677
678     # change flow direction of cold channel if co_flow
679     if M.co_flow:
680         F.mdotC = -F.mdotC
681
682     # print 'T0', T0
683
684     # set up tuple of optional inputs for use by fsolve
685     args = (M, G, F, 0)
686
687     if M.optim == 'fsolve':
688         T , infodict, status, mesg = sci.optimize.fsolve(equations,T0,args=args,
                full_output=1)
689     elif M.optim == 'root:hybr':
690         sol = sci.optimize.root(equations,T0,args=args,method='hybr',options={'
                xtol':1.e-12})
691         status = sol.status
692         T = sol.x
693         mesg = sol.message
694     elif gdata.optim == 'root:lm':
695         sol = sci.optimize.root(equations,A0,args=args,method='lm',options={'eps'
                :1.e-3, 'xtol':1.e-12, 'ftol':1e-12})
696         status = sol.status
697         A = sol.x
698         mesg = sol.message
699     elif gdata.optim == 'root:Newton-CG':
700         sol = sci.optimize.root(equations,A0,args=args,method='lm',options={'eps'
                :1.e-3, 'xtol':1.e-12})
701         status = sol.status
702         A = sol.x
703         mesg = sol.message
704     elif gdata.optim == 'root:df-sane':
705         sol = sci.optimize.root(equations,A0,args=args,method='df-sane',options={'
                ftol':1.e-12})
706         status = sol.status
707         A = sol.x
708         mesg = sol.message
709     else:
```

```
710         raise MyError("gdata.optim = '' not set preoperly.")
711
712     if status is not 1:
713         print mesg
714         raise MyError('HX_solver.py: fsolve unable to converge.')
715
716     TH, TWH, TWC, TC = open_T(T,F.TH_in,F.TC_in,F.PH_in,F.PC_in,M,F)
717                                 # open_T(T,F.TH_in,F.TC_in,M.N_cell)
718
719     # create pressure traces for output
720     PH, PC= equations(T, M, G, F, 2)
721
722     if M.print_flag:
723         print "Plotting results"
724         plot_HX(TH,TWH,TWC,TC,M.N_cell)
725         error, T, Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8= equations(T, M, G, F,  1)
726         plot_HXq(Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, M.N_cell, G.HX_L)
727         plot_Hp(PH,PC,M.N_cell)
728
729         #print Q1, '\n', Q2, '\n', Q3, '\n', Q4, '\n', Q5, '\n', Q6, '\n', Q7 , '\
                n', Q8
730
731         print "\n"
732         print 'Temperatures:'
733         print 'Hot-channel            ' , TH
734         print 'Hot-channel wall temp ' , TWH
735         print 'Cold-channel wall temp' , TWC
736         print 'Cold-channel           ' , TC
737         print '\n'
738         print 'Pressures:'
739         print 'Hot-channel  ' , PH
740         print 'Cold-channel ' , PC
741
742         print "\n \n"
743         print "Power Transferred - (H) channel"
744         print 'T_in     (K): %.2f ' %(TH[0])
745         print 'T_out    (K): %.2f ' %(TH[-1])
746         print 'Delta T (K): %.2f ' %(abs(TH[0]-TH[-1]))
747         P_inH =  CP.PropsSI('HMASS','P',F.PH_in,'T',TH[0],F.fluidH) * F.mdotH
748         P_outH =  CP.PropsSI('HMASS','P',F.PH_out,'T',TH[-1],F.fluidH) * F.mdotH
749         rho_in = CP.PropsSI('D','P',F.PH_in,'T',TH[0],F.fluidH)
750         rho_out = CP.PropsSI('D','P',F.PH_out,'T',TH[-1],F.fluidH)
751         mu_in = CP.PropsSI('V','P',F.PH_in,'T',TH[0],F.fluidH)
752         mu_out = CP.PropsSI('V','P',F.PH_out,'T',TH[-1],F.fluidH)
753         print 'Power    (kW): %.2f ' %((P_inH - P_outH)/1e3)
754         print 'Reynolds number (in) : %.2f ' %( rho_in *F.mdotH/G.AH/rho_in * G.
                Lc_H / mu_in )
755         print 'Reynolds number (out): %.2f ' %( rho_out *F.mdotH/G.AH/rho_out * G.
                Lc_H / mu_out )
756         print "\n"
757         print "Power Transferred - (C) channel"
758         print 'T_in     (K): %.2f ' %(TC[-1])
759         print 'T_out    (K): %.2f ' %(TC[0])
760         print 'Delta T (K): %.2f ' %(abs(TC[0]-TC[-1]))
761         P_inC =  CP.PropsSI('HMASS','P',F.PC_in,'T',TC[0],F.fluidC) * F.mdotC
762         P_outC =  CP.PropsSI('HMASS','P',F.PC_out,'T',TC[-1],F.fluidC) * F.mdotC
763         rho_in = CP.PropsSI('D','P',F.PC_in,'T',TC[0],F.fluidC)
764         rho_out = CP.PropsSI('D','P',F.PC_out,'T',TC[-1],F.fluidC)
765         mu_in = CP.PropsSI('V','P',F.PC_in,'T',TC[0],F.fluidC)
766         mu_out = CP.PropsSI('V','P',F.PC_out,'T',TC[-1],F.fluidC)
767         print 'Power    (kW): %.2f ' %((P_inC - P_outC)/1e3)
```

```
768         print 'Reynolds number (in) : %.2f ' %( rho_in *F.mdotC/G.AC/rho_in * G.
                Lc_C / mu_in )
769         print 'Reynolds number (out): %.2f ' %( rho_out *F.mdotC/G.AC/rho_out * G.
                Lc_C / mu_out )
770         print "\n \n"
771
772         print 'Heat Transfer Info:'
773         DT_A = TH[0] - TC[0]; DT_B = TH[-1] - TC[-1]
774         T_LM = (DT_B - DT_A) / np.log(DT_B/DT_A)
775         #T_LM = abs(((TH[-1]-TC[0]) - (TH[0]-TC[-1])) / (np.log( (TH[-1]-TC[0])/(
                TH[0]-TC[-1]) ) ))
776         print 'Delta T Log Mean (K): %.2f' %( T_LM )
777         print 'HTC  (W /(m K): %.2f' %( abs(P_inC - P_outC) / G.Area / abs(T_LM) )
778
779         print "\n \n"
780         plt.draw()
781
782
783         plt.pause(1) #<-------
784         print '\n \n'
785         raw_input("<Hit Enter To Close Figures>")
786         plt.close()
787
788     PH_out = PH[-1]
789     PC_out = PC[0]
790     TH_out = TH[-1]
791     TC_out = TC[0]
792
793     return PH_out, TH_out, PC_out, TC_out, PH, TH, PC, TC, T0
794 ###
795 ###
796 def plot_HX(TH,TWH,TWC,TC, N_cell):
797     fig = plt.figure()
798     plt.plot(np.linspace(0,1.,num=N_cell+1),TH, '--',label="(H) Channel")
799     plt.plot(np.linspace(0.5/N_cell,1.-0.5/N_cell,num=N_cell),TWH, 'o--',label="
            Wall")
800     plt.plot(np.linspace(0.5/N_cell,1.-0.5/N_cell,num=N_cell),TWC, 'o-',label="Wall
            ")
801     plt.plot(np.linspace(0,1.,num=N_cell+1),TC, label = "(C) Channel")
802     plt.ylabel('Temperature (K)')
803     plt.xlabel('Position along Heat Exchanger (normalised)')
804     plt.title('Temperature Distributions in Heat Exchanger')
805     plt.legend(loc=2)
806 ###
807 ###
808 def plot_Hp(PH,PC, N_cell):
809     fig, ax1 = plt.subplots()
810     l1 = ax1.plot(np.linspace(0,1.,num=N_cell+1),PH, '--',label="Pressure (H)
            Channel")
811     ax2 = ax1.twinx()
812     l2 = ax2.plot(np.linspace(0,1.,num=N_cell+1),PC, label = "Pressure (C) Channel
            ")
813     ax1.set_ylabel('Pressure H channel (Pa)')
814     ax2.set_ylabel('Pressure C channel (Pa)')
815     plt.xlabel('Position along Heat Exchanger (normalised)')
816     plt.title('Pressure Distributions in Heat Exchanger')
817     lines = l1+l2
818     labels = [l.get_label() for l in lines]
819     plt.legend(lines,labels,loc=2)
820
821 ###
822 ###
```

```
823 def plot_HXq(Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, N_cell,Length):
824     fig = plt.figure()
825     plt.plot(np.linspace(0,1.,num=N_cell+1)[0:-1]              ,np.array(Q1)/1.e3,
            label="q1")
826     plt.plot(np.linspace(0,1.,num=N_cell+1)[1:]                ,np.array(Q2)/1.e3,
            label="q2")
827     plt.plot(np.linspace(0.5/N_cell,1.-0.5/N_cell,num=N_cell),np.array(Q3)/1.e3,
            label="q3")
828     plt.plot(np.linspace(0.5/N_cell,1.-0.5/N_cell,num=N_cell),np.array(Q4)/1.e3,
            label="q4")
829     plt.plot(np.linspace(0,1.,num=N_cell+1)[0:-1]              ,np.array(Q5)/1.e3,
            label="q5")
830     plt.plot(np.linspace(0,1.,num=N_cell+1)[1:]                ,np.array(Q6)/1.e3,
            label="q6")
831     plt.plot(np.linspace(0.5/N_cell,1.-0.5/N_cell,num=N_cell),np.array(Q7)/1.e3,
            label="q7")
832     plt.plot(np.linspace(0.5/N_cell,1.-0.5/N_cell,num=N_cell),np.array(Q8)/1.e3,
            label="q8")
833     plt.ylabel('Heat Flow (kW)')
834     plt.xlabel('Position along Heat Exchanger (normalised)')
835     plt.title('Energy Fluxes in Heat Exchanger')
836     plt.legend(loc=2)
837     ###
838     ###
839     fig = plt.figure()
840     plt.plot(np.linspace(0.5/N_cell,1.-0.5/N_cell,num=N_cell),np.array(Q3)/1.e3/(
            Length/N_cell),label="q3")
841     plt.plot(np.linspace(0.5/N_cell,1.-0.5/N_cell,num=N_cell),np.array(Q4)/1.e3/(
            Length/N_cell),label="q4")
842     plt.plot(np.linspace(0.5/N_cell,1.-0.5/N_cell,num=N_cell),np.array(Q7)/1.e3/(
            Length/N_cell),label="q7")
843     plt.plot(np.linspace(0.5/N_cell,1.-0.5/N_cell,num=N_cell),np.array(Q8)/1.e3/(
            Length/N_cell),label="q8")
844     plt.ylabel('Flow per unit length (kW/m)')
845     plt.xlabel('Position along Heat Exchanger (normalised)')
846     plt.title('Energy Fluxes in Heat Exchanger')
847     plt.legend(loc=4)
848 ###
849 ###
850 shortOptions = ""
851 longOptions = ["help", "job=", "noprint"]
852 ###
853 ###
854 def printUsage():
855     print ""
856     print "Usage: HX_solver.py [--help] [--job=<jobFileName>] [--noprint]"
857     print "\n"
858     print " --help       Dispay help"
859     print "\n"
860     print " --job=       Use this to specify the job file."
861     print "\n"
862     print " --noprint    This suppresses on screen outputs."
863     return
864 ###
865 ###
866 class MyError(Exception):
867     def __init__(self, value):
868         self.value = value
869     def __str__(self):
870         return repr(self.value)
871 ###
872 ###
```

```
873 if __name__ == "__main__":
874     userOptions = getopt(sys.argv[1:], shortOptions, longOptions)
875     uoDict = dict(userOptions[0])
876 
877     if len(userOptions[0]) == 0 or uoDict.has_key("--help"):
878         printUsage()
879         sys.exit(1)
880 
881     # execute the code
882     try:
883         main(uoDict)
884         print "\n \n"
885         print "SUCESS."
886         print "\n \n"
887 
888     except MyError as e:
889         print "This run has gone bad."
890         print e.value
891         sys.exit(1)
```