University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: MSc in Petroleum Engineering – Well and Drilling Engineering	Spring semester, 2020 Open
Author: Juan Camilo Gonzalez Angarita	
Supervisor(s): UiS - Mesfin Belayneh External supervisor: Pro Well Plan AS – Eirik Lyngvi	
Title of master's thesis: Integrated Modelling and Simulation of Wellbore Heat Transfer Processes through High-level Programming, Sensitivity Analysis and Initial Approach with Machine Learning Predictive Models	
Credits: 30	
Keywords: Well temperature model, Temperature distribution, Heat transfer simulation, Python, Predictive models, Machine Learning	Number of pages: 106 + supplemental material/other: 112 Stavanger, 30th June 2020

Juan Camilo Gonzalez Angarita

Integrated Modelling and Simulation of Wellbore Heat Transfer Processes through High-level Programming, Sensitivity Analysis and Initial Approach with Machine Learning Predictive Models

Master Thesis Project for the degree of

MSc in Petroleum Engineering

Stavanger, June 2020

University of Stavanger

Faculty of Science and Technology

Department of Energy and Petroleum Engineering

Abstract

During designing of downhole systems and selecting equipment and materials, engineers must consider in-situ conditions before taking decisions in order to be able to handle any operation in a safe and adequate manner. The well temperature profile is mainly imposed by the formation temperature; however, this can vary significantly during operations in different ways. Several properties of fluids and pipes would take part of the heat transfer process such as flow rates, specific heat capacities, thermal conductivities, densities and viscosities.

This work gathers and implements various complementary models to simulate the heat transfer across the wellbore during drilling, production and injection. The entire wellbore is discretized, and the models are solved numerically by applying the Crank-Nicholson finite differences method for two dimensions. All the calculations are programmed in python and are released as an open source repository. Besides, a sensitivity analysis is performed for the three main operations (drilling, production and injection), describing individual effects of the parameters on the temperature variation.

In addition, prediction models are developed from true and simulated data. These are presented in detail from the data acquisition up to the model assessment. Thus, their performances are compared with the physics-based models, where accuracy, simplicity and computing time play a key role within the engineering tasks; specially when analyzing numerous wells and/or conditions.

Acknowledgements

At first, I would like to thank the professor Mesfin Belayneh of the Department of Energy and Petroleum Technology at the University of Stavanger. Under his supervision, he offered to me all his knowledge, expertise and support that allowed to complete this project successfully.

To Pro Well Plan for supporting completely every single step of the project, without their knowledge and experience it would not have been possible. Specially to Eirik Lyngvi, Khushal Adlakha and Magnus Tvedt.

Finally, I would like to thank my family and friends, for the support. Specially to my grandpa Edmundo Angarita, who is always blessing me, now from heaven.

List of Symbols

Symbol	Description	Units
A_n	Area of nuzzles	m^2
A	Azimuth	$\circ$
$B_{i,j}$	Constant values at cell i, j	$dimensionless$
B_y	Buoyancy factor	$dimensionless$
C	Center	$cell$
c	Specific heat capacity	$J/(kg \cdot °C)$
D	Diameter	m
D_H	Hydraulic Diameter	m
E	East	$cell$
e	Eccentricity	$dimensionless$
e_{avg}	Average eccentricity	$dimensionless$
f	Friction factor	$dimensionless$
F_n	Net normal force	N
F	Force	N
g	Gravitational acceleration	$kg \cdot m/s^2$
g_c	Unit conversion factor	$m \cdot s^2/h^2$
Gr	Grashof number	$dimensionless$
h	Convective heat transfer coefficient	$W/(m^2 \cdot °C)$
I	Inclination	$\circ$
J	Joule's constant	Nm/cal
K	Consistency index	$dimensionless$
L	Length	m
M	Torque	$N \cdot m$
N	North	$cell$
n	Flow behavior index	$dimensionless$
Nu	Nusselt number	$dimensionless$
P	Pressure	Pa
P_s	Pressure at surface	Pa
Pr	Prandtl number	$dimensionless$
Q	Heat source term	J

q	Flow rate	m^3/h
r	Radius	m
r_δ	Annular clearance	m
Ra	Rayleigh number	$dimensionless$
Re	Reynold number	$dimensionless$
R_{ecc}	Ratio of pressure drop by eccentricity	$dimensionless$
R_{rot}	Ratio of pressure drop by rotation	$dimensionless$
S	South	$cell$
S_{avg}	Average amplitude of the inner pipe	m
S_o	Amplitude of the inner pipe	m
T	Temperature	$°C$
T_{bit}	Torque on bit	$kN \cdot m$
$T_{i,j}^n$	Temperature at time n at cell i, j	$°C$
T_s	Temperature at surface	$°C$
U	Velocity	m/s
v	Fluid velocity	m/s
W	West	$cell$
W_d	Weight of drill pipe in air	N
W_{eff}	Effective weight	kN
ΔE	Change in East	m
ΔM	Change in torque	$N \cdot m$
ΔN	Change in North	m
α	Thermal expansion coefficient	$1/°C$
β	Isothermal bulk modulus	Pa
η	Drill bit efficiency	$dimensionless$
λ	Thermal conductivity	$W/(m \cdot °C)$
ω	Rotary speed	$rotation/s$
μ_{app}	Apparent viscosity	$Pa \cdot s$
μ_t	Sliding friction coefficient between DP-wellbore	$dimensionless$
ρ_0	Density at surface conditions	kg/m^3
τ_0	Yield stress	Pa
τ_w	Wall shear stress	Pa
$\bar{\theta}$	Average inclination angle of element	rad

$\Delta\alpha$	Increase in azimuth over length of element	rad
τ	Shear stress	Pa
γ	Shear rate	s^{-1}
ρ	Density	kg/m^3

List of Abbreviations

Symbol	Description	Units
DL	Dogleg	$^\circ$
MD	Measured depth	m
RF	Ratio factor	$dimensionless$
ROP	Rate of Penetration	m/h
TVD	True vertical depth	m
WOB	Weight on bit	kN

List of Figure

List of Tables

Table of Contents

1. Introduction

An accurate estimation of well temperature distributions is very important for different vital tasks during the well planning process. Either for regular and deeper wells the engineer needs to consider this information in order to select appropriate equipment and materials, also to analyze properly the stresses along the tubulars. Moreover, the knowledge of the temperature profile facilitates in handling more situations such as drilling fluid formulation (Adewole & Najimu, 2017; Zhao et al., 2018), cementing program design (Alvi et al., 2020) and design of geothermal wells, improving the energy extraction process.

Throughout this thesis is presented the implementation and integration of different temperature models in one single python-based software tool. The wellbore heat transfer process is described and simulated for drilling, production and injection operations. Besides, a sensitivity analysis is performed separately to get a better understanding of physics involved and data analysis methods are applied to analyze a data-driven approach using data from the North Sea.

This work describes different physics involved in heat transfer calculations for the entire well, all the equations and models are based and fully integrated within an open source project available on GitHub. The entire processes and developed tools are described and used to generate results, comparisons and finally conclusions.

1.1. Objective

The primary objective of the thesis work is to develop tools for an "Integrated Modelling and Simulation of Wellbore Heat Transfer Processes through High-level Programming, Sensitivity Analysis and Initial Approach with Machine Learning Predictive Models".

1.2. Research program

As shown in Figure 1, the research method comprises of three main parts. The first part deals with the wellbore temperature modelling under drilling, injection and production operations. During modelling, all the possible heat transfer mechanisms in the axial and lateral directions along with the energies source terms are included. The models are solved with Crank

Nicholson numerical schemes. The accuracy of the model was verified with the measured literature data and commercial software (Landmark) under the considered experimental and simulation setups. The second part involves a sensitivity analysis of the parameters, conducted to verify the degree of the impact as well as the tendency of the results if it makes sense or not. Finally, a machine learning approach is performed for modelling of temperature profile variations under operation conditions, and formation temperature based on several North Sea wells data.

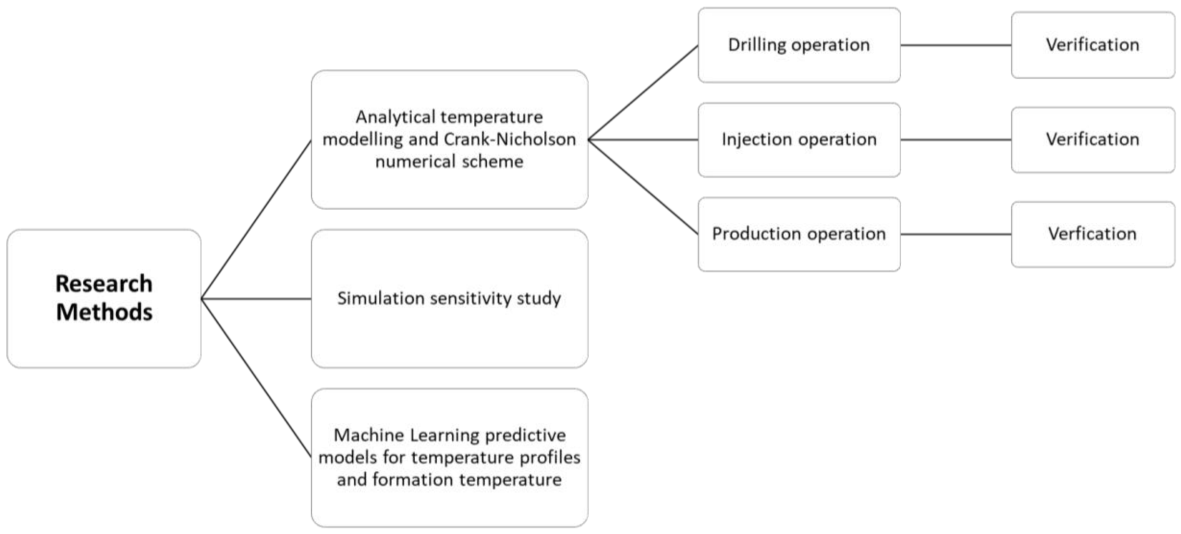

Research Methods
Analytical temperature modelling and Crank-Nicholson numerical scheme
Simulation sensitivity study
Machine Learning predictive models for temperature profiles and formation temperature
Drilling operation
Injection operation
Production operation
Verification
Verification
Verfication

Figure 1. Overview of the research program

2. Heat Transfer Theory

Thermal energy transfer occurs when there is a difference in temperature within a system and the internal energy of the parts changes according to the first law of thermodynamics. The heat transfer always happens from a system of higher temperature to another system of lower temperature. This energy exchange is generally denoted as $Q\ [W]$ and could take place through three possible modes: conduction, convection and radiation.

2.1. Heat Transfer Models

Several methods, models and approaches have been analyzed during decades, it is generally difficult to predict the well temperature distribution under many dynamic conditions involved during typical operations such as drilling, production and injection.

During the sixties, a methodology was proposed to predict the well temperature profile (Edwardson et al., 1962); and the classic model for temperature estimation of fluids, tubing and casing along the wellpath was presented (Ramey, 1962), introducing total heat transfer coefficient and a time function. Besides, the model assumed steady-state heat transfer within wellbore and transient behavior across the formation; and neglects thermal resistance of tubulars, frictional and kinetic energy effects.

Furthermore, a wellbore temperature estimation procedure was presented for an injection operation case with a gas filled annulus (Willhite, 1967), considering axial and radial conduction, radiation and natural convection as involved heat transfer mechanisms. In addition, a general definition for the overall heat transfer coefficient for wellbores was introduced. Besides, a numerical solution approach was implemented under steady and pseudo-steady states, through the finite element method and deriving some energy equations for the fluid in drill pipe (Raymond, 1969), annulus and surroundings.

Later, an analytical model under steady state was introduced to predict the well temperature profile (Holmes & Swift, 1970). Then, some energy equations were presented (Keller et al., 1973), considering the wellbore geometry and including casings and cement sheaths.

Moreover, a different approach was implemented to estimate the effect of circulating pressure losses on temperature along the wellbore during drilling (Marshall & Bentsen, 1982).

During the nineties, a method was presented to estimate the Joule-Thomson coefficient (Alves et al., 1992), as result of the pressure losses while circulating upwards through the tubing. Also, a steady-state two-phase flow model was introduced for production operations (A. R. Hasan & Kabir, 1994), including a new approach for transient behavior across the formation. Also, an inverse circulation model was stablished (Kabir et al., 1996).

Additionally, an analytical model was developed under steady state to estimate the well temperature distribution (A. Rashid Hasan et al., 2009), considering inclination, different geothermal gradients and the Joule-Thomson effect.

2.2. Heat Transfer Mechanisms

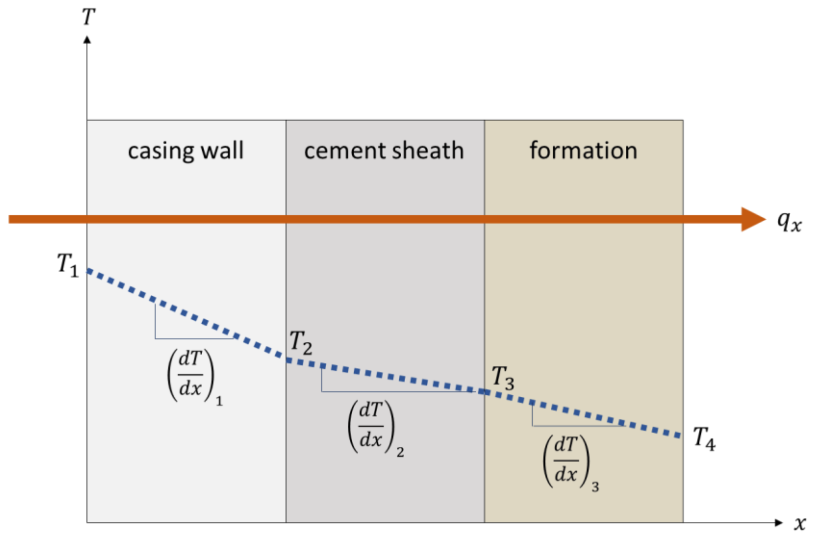
2.2.1. Conduction

This mode of energy transfer is defined as "heat due to temperature difference within a body or between bodies in thermal contact without the involvement of mass flow and mixing" (Kothandaraman, 2006). In other words, the direct contact between solids or fluids with a difference in temperature leads to heat transfer by conduction. Particles with higher kinetic or vibrational energy will transfer it to the nearby ones with lower energy.

$$Q_x = -\lambda \cdot A \cdot \frac{dT}{dx} \tag{1}$$

$where$:

λ: $thermal\ conductivity$



T
casing wall
cement sheath
formation
q_x
T_1
T_2
T_3
T_4
$\left(\frac{dT}{dx}\right)_1$
$\left(\frac{dT}{dx}\right)_2$
$\left(\frac{dT}{dx}\right)_3$
x

Figure 2. Heat conduction in radial direction

2.2.2. Convection

This mode of heat transfer "is basically conduction in a very thin fluid layer at the surface and then mixing caused by the flow" (Kothandaraman, 2006). Thus, convection takes place between two surfaces by a fluid in motion through molecular interaction. Diffusion and advection are the heat transfer mechanisms involved in this process. Two types of convection are considered: natural convection and forced convection.

$$Q = h \cdot A \cdot \Delta T \tag{2}$$

$where$:

h: $convective\ heat\ transfer\ coefficient$

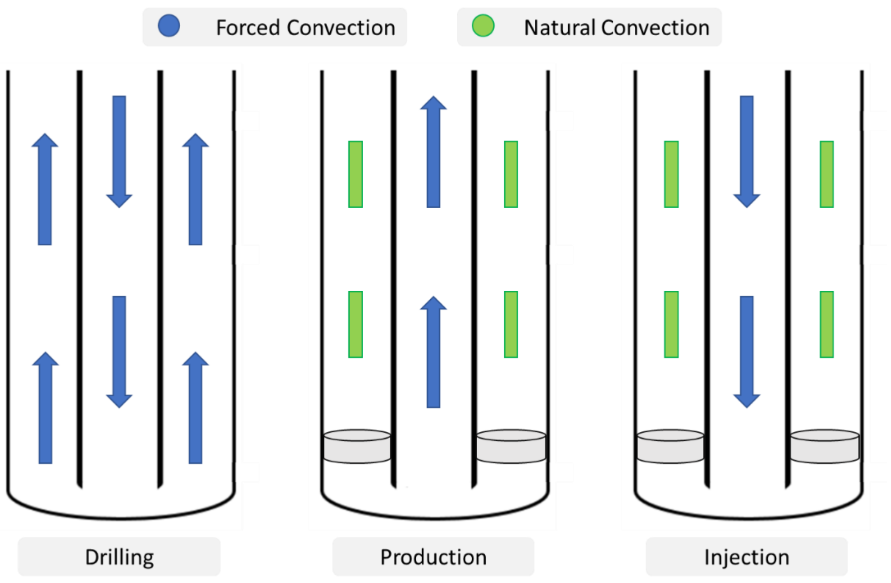


Figure 3. Heat convection during drilling, production and injection

- *Natural convection*

This type of convective heat transfer occurs when the change in temperature of the fluid causes a change in density, generating a buoyancy effect and internal movement across the fluid.

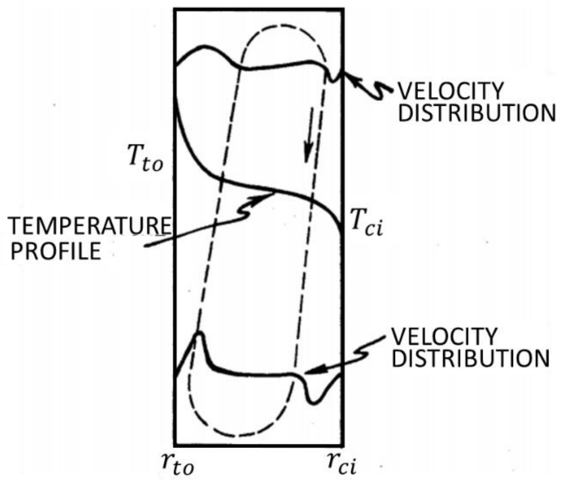


Figure 4. Natural convection in casing annulus (taken from(Willhite, 1967))

- *Forced convection*

This case assumes an external force acting over the fluid and producing its circulation. In a wellbore, such forces come from natural or artificial (pumping) pressure gradients. There is forced convection acting during drilling inside the drill pipe and in the annular, and during production inside the tubing (upwards), similarly as during injection (downwards).

2.2.3. Radiation

Heat radiation implies the energy transfer through electromagnetic radiation. This is known as thermal radiation when happens due to a temperature gradient between surfaces. In the case of thermal radiation reaching a surface, this is distributed depending on the material properties of that surface, thus, some of the energy is reflected, some is absorbed within the material and the remaining is transmitted. "No medium is required for radiative transfer but the surfaces should be in visual contact for direct radiation transfer" (Kothandaraman, 2006).

$$Q = F \cdot \sigma \cdot A \cdot \Delta(T^4) \tag{3}$$

$where$:

$F: factor\ depending\ on\ geometry\ and\ surface\ properties$

$\sigma: Stefan\ Boltzmann\ constant\ 5.67 \times 10^{-8}\ \frac{W}{m^2K^4}$

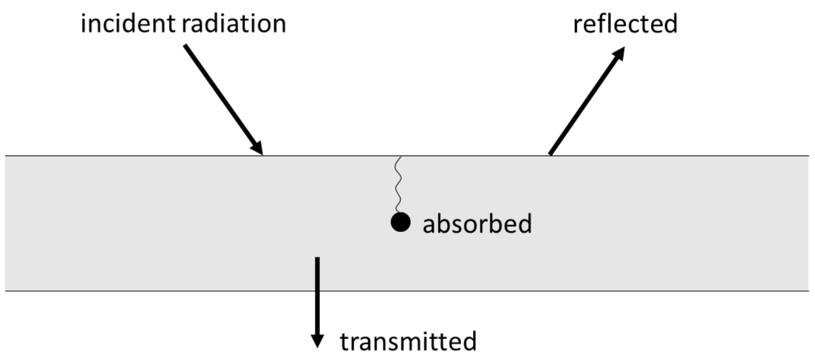


Figure 5. Effect of incident radiation (taken from Holman, 2010)

3. Temperature Modelling and Verification

The entire well is discretized defining properly the involved sections in the 2D system. The Figure 6 shows the defined elements along the X-axis, where the surrounding space is considered as one single section from the first cement sheath up to the last one.

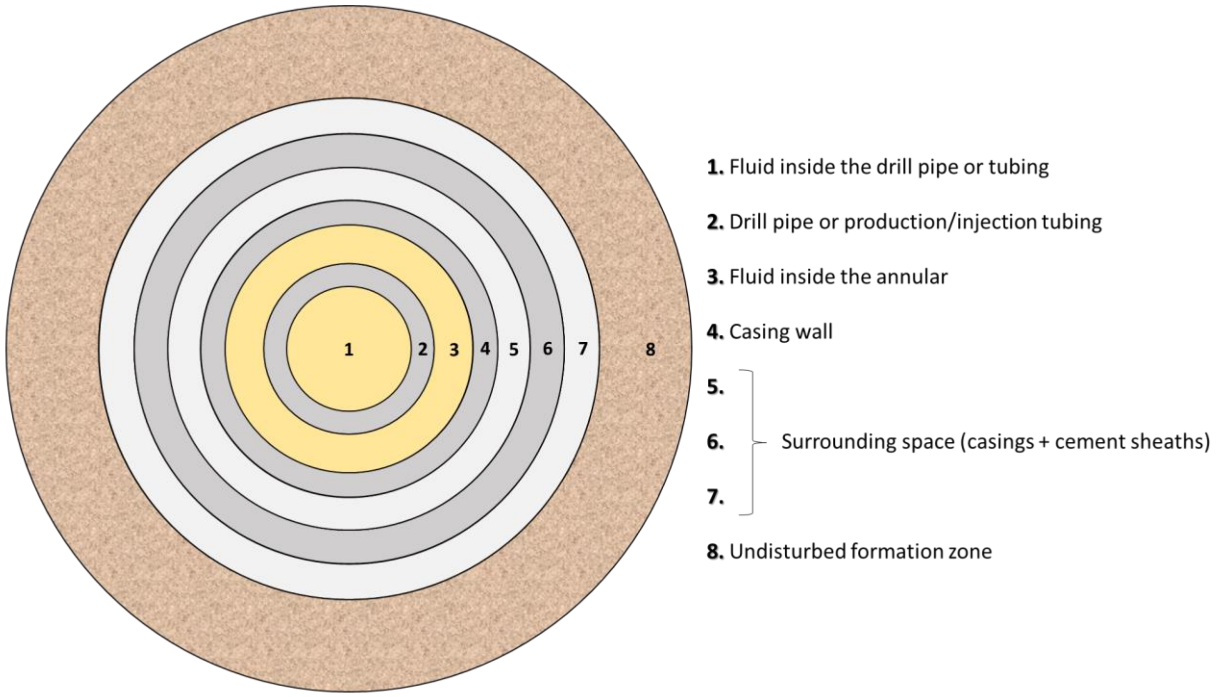

1
2
3
4
5
6
7
8
1. Fluid inside the drill pipe or tubing
2. Drill pipe or production/injection tubing
3. Fluid inside the annular
4. Casing wall
5.
6. Surrounding space (casings + cement sheaths)
7.
8. Undisturbed formation zone

Figure 6. Wellbore radial scheme

In the following, the method of discretization and interaction of the codes are briefly summarized.

The individual analysis for each cell requires an identification system able to normalize the process from surface to the bottom. The figure below shows the 2D tag system implemented to the correlations between these cells. North (N), West (W), Center (C), East (E) and South (S) would be the coefficients of the positions $(i, j-1), (i-1, j), (i, j), (i+1, j)$ and $(i, j+1)$ respectively.

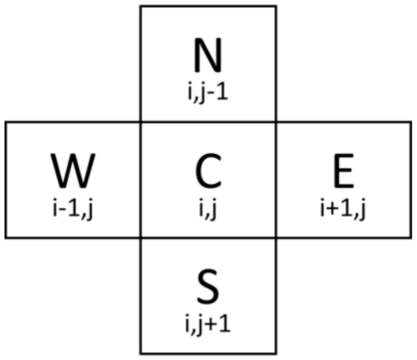


Figure 7. Grid cell scheme

Each grid cell generates an equation in which is related with the other nearby cells; thus, a penta-diagonal matrix is generated, and the system of equations is solved by applying the form $Ax = B$. In this way, A represents all the coefficients involved in the equations, B the vector of constant values and x the temperature value of for each grid cell.

$$N_{i,j}T_{i,j-1}^{n+1} + W_{i,j}T_{i-1,j}^{n+1} + C_{i,j}T_{i,j}^{n+1} + E_{i,j}T_{i+1,j}^{n+1} + S_{i,j}T_{i,j+1}^{n+1} = B_{i,j} \tag{4}$$

$$\begin{bmatrix} C & E & & \cdots & S & & & & \\ W & C & E & & \cdots & S & & & \\ & W & C & E & & & \cdots & & \\ \cdots & & \cdots & \cdots & \cdots & & \cdots & S & \\ N & \cdots & & W & C & E & & \cdots & S \\ & N & \cdots & & W & C & E & & \cdots \\ & & \cdots & & & \cdots & \cdots & \cdots & \\ & & & N & \cdots & & W & C & E \\ & & & & N & \cdots & & W & C \end{bmatrix} \begin{bmatrix} T_{1,1}^{n+1} \\ T_{2,1}^{n+1} \\ \vdots \\ T_{i-1,j}^{n+1} \\ T_{i,j}^{n+1} \\ \vdots \\ T_{i_{max}-1,j_{max}}^{n+1} \\ T_{i_{max},j_{max}}^{n+1} \end{bmatrix} = \begin{bmatrix} B_{1,1} \\ B_{2,1} \\ \vdots \\ B_{i-1,j} \\ B_{i,j} \\ \vdots \\ B_{i_{max}-1,j_{max}} \\ T_{i_{max},j_{max}}^{n+1} \end{bmatrix}$$

The calculation tool is compound by several python scripts involving different functions that are in charge of specific tasks such as gather all the inputs requested to perform the calculation (input.py), generate the wellpath object (wellpath.py), determine the initial conditions (initcond.py), calculate all the heat transfer coefficients (heatcoefficients.py) and build the penta-diagonal matrix and solve the system of equations (linearsystem.py). Furthermore, one of them is in charge of connecting the functions and create a loop increasing the timesteps (main.py) considering to update the fluid density from temperature changes for each time step (fluid.py) and the torque and drag forces (torque_drag.py) if the calculation is done for a drilling operation (*).

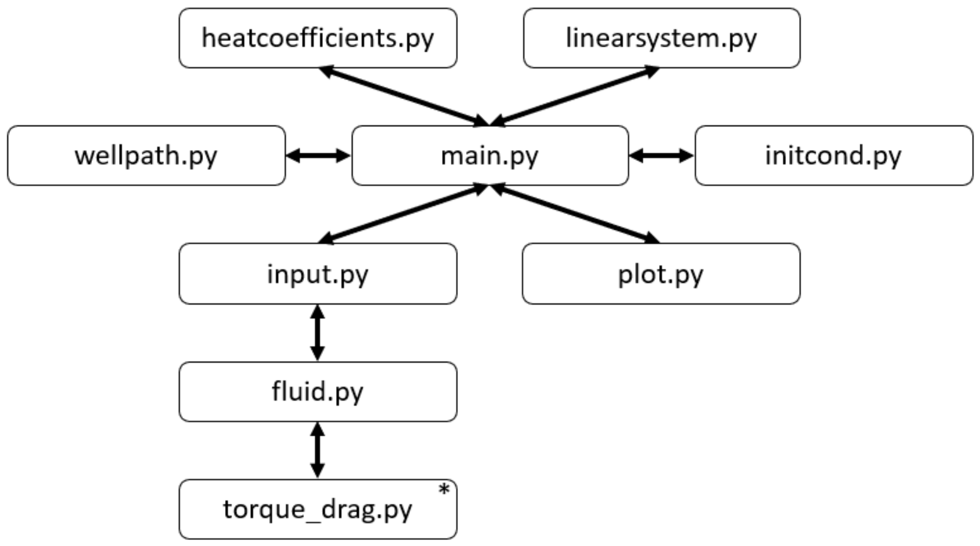


Figure 8. Diagram of interaction between code scripts

In other words, a hydraulics model in “fluid.py” is implemented to calculate the pressure losses and ECD in order to update temperature and pressure dependent fluid parameters, which are input to simulate the temperature models of drilling, injection and production modules. Also, these parameters are used in “torque_drag.py” to calculate the torque and drag for drilling operation, allowing to estimate the temperature profile and then update the density and viscosity profiles of the fluid for further prediction.

The entire process starts by setting inputs as shown in the Figure 9. These parameters allow to generate the entire wellpath and define the initial conditions. The initial temperature profile defines the fluid density profile at the current time step, if it is a drilling operation, the torque and drag forces are calculated before generating the respective heat coefficient for each grid cell and solve the produced system of equations. If the final time step has not been reached, the calculated temperature profile is now the initial condition and the entire process in repeated.

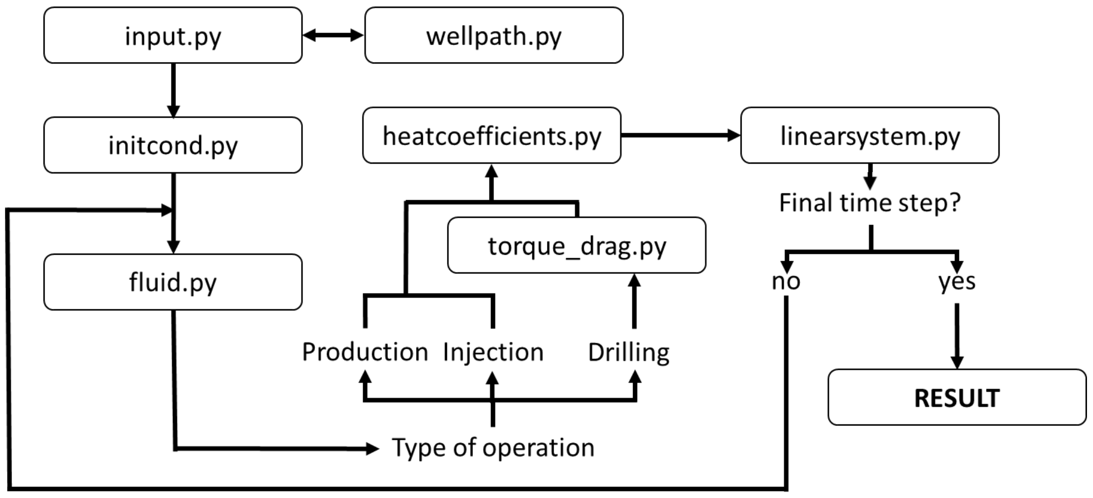


Figure 9. General software flowchart

Density of fluids are sensitive to changes in temperature and pressure. An increase of temperature will cause an expansion of the fluid decreasing the density. On the other hand, the fluid is compressed when increasing the pressure and thus density increases as displayed in Figure 10. A linearized equation of state allows to calculate the fluid density as a function of temperature and pressure (Stamnes, 2011).

$$\rho = \rho_s + \frac{\rho_s}{\beta}(P - P_s) - \rho_0 \alpha (T - T_s) \tag{5}$$

$where$:

ρ_s, P_s and T_s are $density, pressure$ and $temperature$ at $surface$ $respectively.$

β: $isothermal$ $bulk$ $modulus$

α: $thermal$ $expansion$ $coefficient$

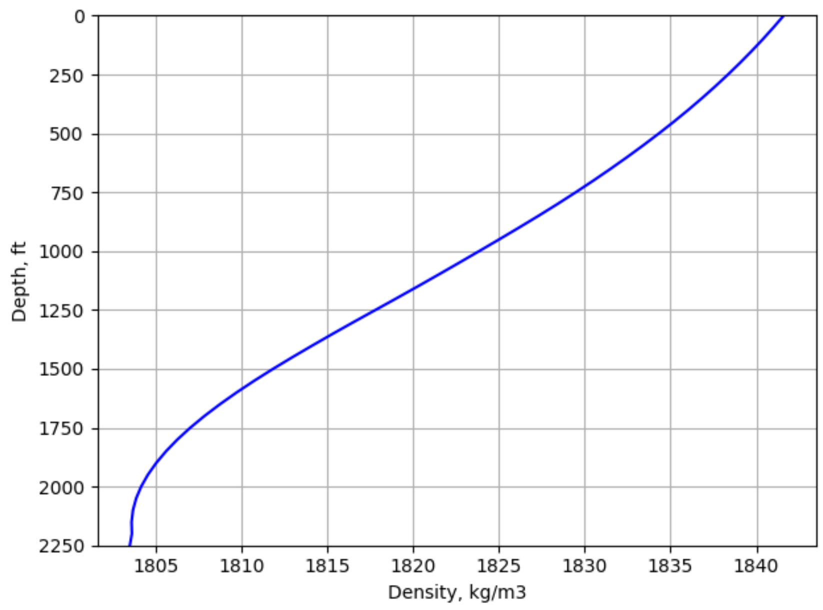


Figure 10. Fluid density profile along the well

The pressure gradient $\left(\frac{dP}{dx}\right)$ through the pipe involves the hydrostatic pressure gradient $\left(\frac{dP}{dx}\right)_h$, frictional pressure gradient $\left(\frac{dP}{dx}\right)_f$ and acceleration pressure gradient $\left(\frac{dP}{dx}\right)_a$ (Sanni, 2018). However, the acceleration gradient is not included in the calculations performed in this thesis.

$$\left(\frac{dP}{dx}\right) = \left(\frac{dP}{dx}\right)_h + \left(\frac{dP}{dx}\right)_f + \left(\frac{dP}{dx}\right)_a \tag{6}$$

<u>Hydrostatic:</u>

$$\left(\frac{dP}{dx}\right)_h = \rho \cdot \mathrm{g} \cdot \mathrm{TVD} \tag{7}$$

<u>Frictional:</u>

$$\left(\frac{dP}{dx}\right)_f = \frac{f\rho \mathrm{L} U^2}{2D} \tag{8}$$

<u>Acceleration:</u>

$$\left(\frac{dP}{dx}\right)_a = -\rho \mathrm{U}\frac{dU}{dx} \tag{9}$$

$where$:	g: gravitational acceleration	L: length
	TVD: true vertical depth	U: flow velocity
	f: friction factor	D: diameter

Furthermore, the fluid viscosities are also affected by changes in temperature and pressure. Figure 11 shows the viscosity profiles through depth when a certain operation is taking place. Several models have been presented in literature considering the type of fluid and specific conditions that are intrinsically related to the operation being performed. This thesis implements three different viscosity models in order to get a better individual approach for drilling, production and injection.

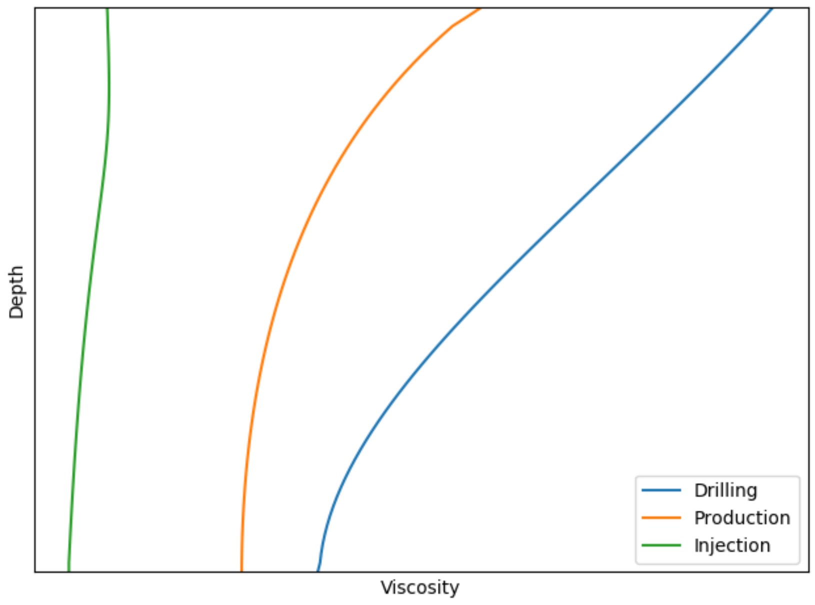


Figure 11. Fluid viscosity profile comparison during drilling, production and injection.

The American Petroleum Institute (API) recommended the following two formulas to estimate the effective viscosity of drilling fluids at lower shear rates at high temperature and pressure. However, since the pressure has little effect compared to the temperature, a simplified version of the equations was introduced (Huang et al., 2020) considering a constant pressure value of 1000 psi.

$$\mu = \mu_0 \cdot e^{(aT+bP)} \tag{10}$$

$where$:	μ: viscosity	μ_0: viscosity at standard condition
	a, b: constants	P: pressure
	T: temperature	

For production, engineers usually implement empirical correlations based on the type of fluids produced within a certain region. Therefore, a formula introduced for North Sea oils (Glaso, 1980) is implemented in this work to capture the relation between dead oil viscosity, API gravity and temperature.

$$\mu = c(\log(\gamma_{API}))^d \tag{11}$$

$$c = 3.141(10^{10})T^{-3.444} \tag{12}$$

$$d = 10.313(\log(T)) - 36.447 \tag{13}$$

$where$:	μ: viscosity	γ_{API}: API gravity
	T: $temperature$	

Moreover, the injection operation should consider the type of fluid since this should change drastically the behavior of the physic properties. This thesis uses a viscosity correlation developed for liquids (Reid et al., 1987).

$$\mu = a \cdot e^{\left(\frac{b}{T}+cT+dT^2\right)} \tag{14}$$

$where$:	μ: viscosity	γ_{API}: API gravity
	a, b, c, d: constants	T: $temperature$

3.1. Wellpath builder

Different methods have been developed to calculate a well course from azimuthal and inclination data considering depth values; such as tangential model, angle average model, balanced tangential model, radius of curvature model, minimum curvature model and quadratic model.

In 1985, the Minimum Curvature method was recognized by the industry as one of the most accurate methods (Amorin & Broni-Bediako, 2010). Equations below show how to calculate dogleg, change in North, change in East and change in True Vertical Depth respectively.

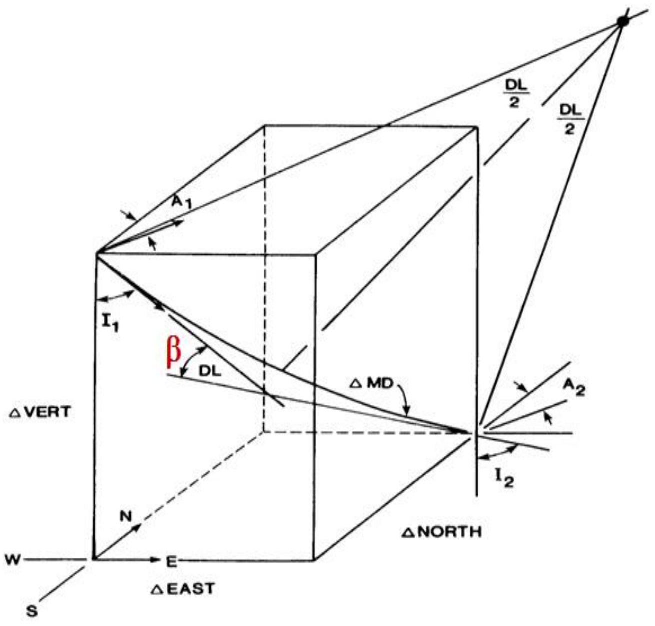


Figure 12. Illustration of minimum curvature method between two survey points (I1, A1) and (I2, A2)(taken from (Langaker & Edvardsen, 2015)).

Dogleg calculation:

$$DL = \cos^{-1}[\cos(I_1)\cos(I_2) - \sin(I_1)\sin(I_2)[1 - \cos(A_2 - A_1)]] \tag{15}$$

Northing calculation:

$$\Delta N = 0.5 \cdot \Delta MD[\sin(I_1) \cdot \cos(A_1) + \sin(I_2) \cdot \cos(A_2)] \cdot RF \tag{16}$$

Easting calculation:

$$\Delta E = 0.5 \cdot \Delta MD[\sin(I_1) \cdot \sin(A_1) + \sin(I_2) \cdot \sin(A_2)] \cdot RF \tag{17}$$

Vertical depth calculation:

$$\Delta TVD = 0.5 \cdot \Delta MD[\cos(I_1) \cdot \cos(I_2)] \cdot RF \tag{18}$$

$$RF = \frac{tan(DL/2)}{(DL/2)} \tag{19}$$

$where$: I_1: first element inclination A_1: first element azimuth

I_2: second element inclination A_2: second element azimuth

The developed python package includes two main ways to generate a wellpath. The first one allows the user to load an existing wellpath by introducing the key data. The other way can create a new well profile by setting some parameters such as type of well, depth of target, kick off point, build angle, end of build, start of drop and end of drop.

Five types of wells are covered in the code: vertical, J-type, S-type, horizontal single curve and horizontal double curve.

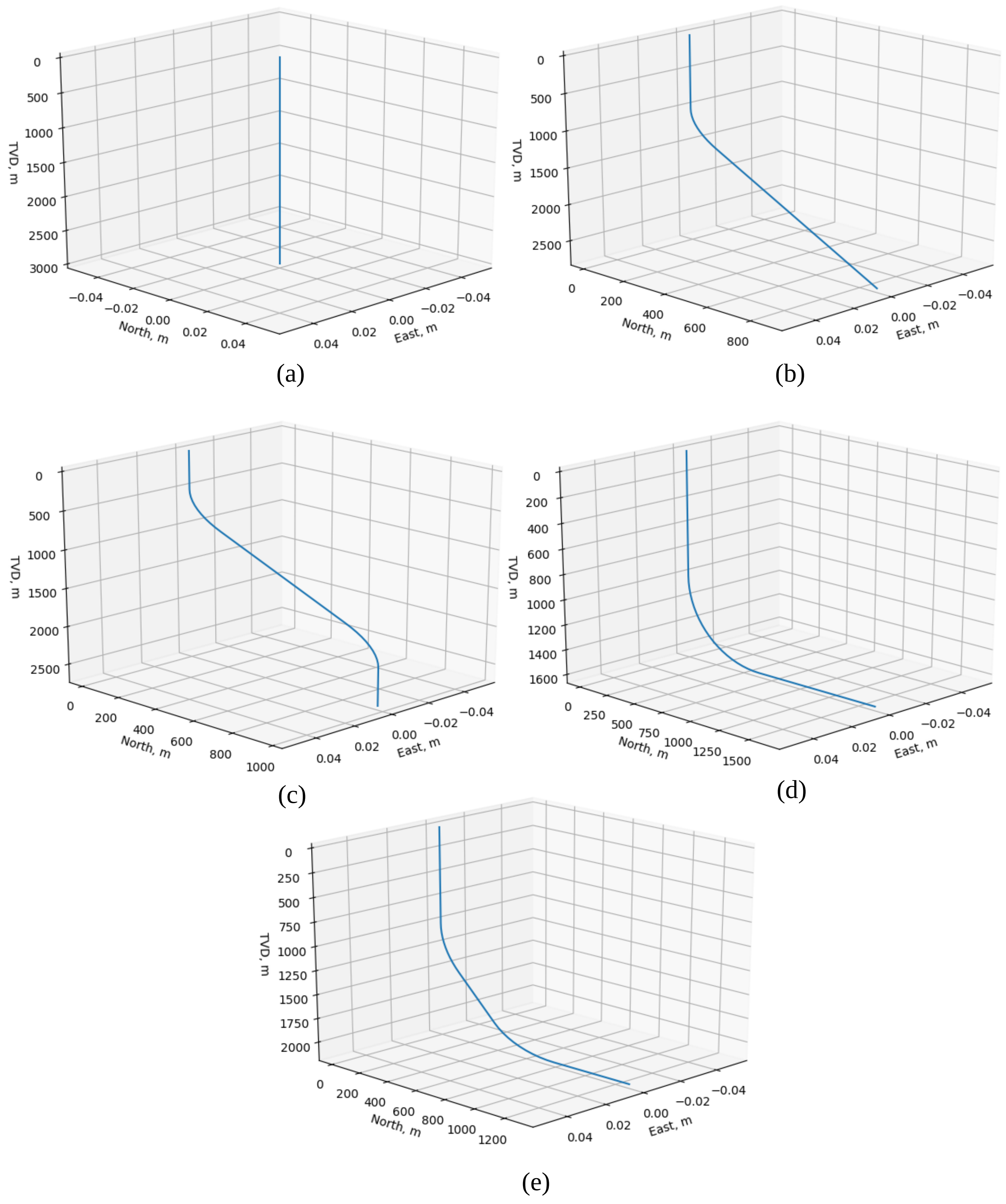


Figure 13. Well profiles. (a) Vertical, (b) J-type, (c) S-type, (d) Horizontal single curve, (e) Horizontal double curve

3.2. Drilling

A drilling process involves several physics events that take important place in the heat transfer phenomenon along the whole wellbore. This model is mainly focused on heat conduction in the radial direction (involves contact of drill pipe, drilling fluid, casings, cement and formation) and forced heat convection due to the mud circulation through the drill string from surface to the drill bit, then upwards through the annular space. Heat source terms due to drill pipe rotation, hydraulic and mechanical frictions are also included.

The heat transfer model was adapted and implemented from aspects of different references (Chang et al., 2018; Marshall & Bentsen, 1982; Wooley, 1980; Zhang et al., 2018).

Some assumptions are involved in the model:

1. The wellbore is a regular cylinder and no buckling segments presented along the well.
2. The specific heat capacity, and thermal conductivity of the fluid are constant.
3. Heat transfer in axial and radial directions.

a. *Fluid Inside Drill Pipe*

During the mud circulation the elements suffer energy changes due to heat transfer within the drilling fluid because of the flow downwards in the axial direction, heat convection between the fluid and the inner wall of the pipe, and heat generated from frictional pressure losses.

$$-\rho_{fp} c_f v_p \frac{\partial T_{fp}}{\partial z} + \frac{2h_{pi}\left(T_p - T_{fp}\right)}{r_{pi}} + \frac{Q_p}{\pi r_{pi}^2} = \rho_{fp} c_{fp} \frac{\partial T_{fp}}{\partial t} \qquad (20)$$

where:

ρ_{fp}: fluid density (inside pipe)	v_p: fluid velocity in pipe
c_f: *fluid heat capacity*	h_{pi}: convective heat coefficient for *inner face of pipe*

$r_{pi}: pipe\ inner\ radius$ $T_p: drill\ pipe\ wall\ temperature$

$T_{fp}: fluid\ temperature\ in\ drill\ pipe$ $Q_p: heat\ source\ term$

$\frac{\partial T_{fp}}{\partial z}: temperature\ diff.in\ depth$ $\frac{\partial T_{fp}}{\partial t}: temperature\ diff.in\ time$

b. *Drill String Wall*

The heat balance along the drill string wall involves heat conduction in the axial direction, and heat convection between the inner wall of the pipe and the fluid inside.

$$\frac{\partial}{\partial z}\left(\lambda_p \frac{\partial T_p}{\partial z}\right) + \frac{2r_{po}h_{po}\left(T_a - T_p\right)}{\left(r_{po}^2 - r_{pi}^2\right)} + \frac{2r_{pi}h_{pi}\left(T_{fp} - T_p\right)}{\left(r_{po}^2 - r_{pi}^2\right)} = \rho_p c_p \frac{\partial T_p}{\partial t} \tag{21}$$

$where$: λ_p: drill pipe thermal conductivity h_{po}: convective heat coefficient for $outer\ face\ of\ pipe$

$T_a: Annulus\ temperature$ $r_{po}: pipe\ outer\ raidus$

c. *Fluid Inside Annular*

The heat balance in the annular section involves the heat transfer due to the flow upwards in the axial direction, heat convection between the fluid and first casing or wellbore face, and heat convection between the fluid and the outer wall of the drill string.

$$\rho_{fa} c_{fa} v_a \frac{\partial T_a}{\partial z} + \frac{2r_{ci}h_{ci}(T_c - T_a)}{\left(r_{ci}^2 - r_{po}^2\right)} + \frac{2r_{po}h_{po}\left(T_p - T_a\right)}{\left(r_{ci}^2 - r_{po}^2\right)} + \frac{Q_a}{\pi\left(r_{ci}^2 - r_{po}^2\right)} = \rho_{fa} c_{fa} \frac{\partial T_a}{\partial t} \tag{22}$$

d. *First Casing*

For this case, the balance equation involves heat conduction in the axial direction, heat transfer with the first layer of cement sheath and heat convection with the drilling fluid in the annulus.

$$\frac{\partial}{\partial z}\left(\lambda_c \frac{\partial T_c}{\partial z}\right) + \frac{2\lambda_{csr}(T_{sr} - T_c)}{(r_{co}^2 - r_{ci}^2)} + \frac{2r_{ci}h_{ci}(T_a - T_c)}{(r_{co}^2 - r_{ci}^2)} = \rho_c c_c \frac{\partial T_c}{\partial t} \tag{23}$$

e. *Surrounding Space*

The heat balance in this comprehensive section involves heat conduction in both axial and radial directions.

$$\frac{\partial}{\partial z}\left(\lambda_{sr} \frac{\partial T_{sr}}{\partial z}\right) + \frac{\partial}{\partial r}\left(\lambda_{srfm} r \frac{\partial T}{\partial r}\right) = \rho_{sr} c_{sr} \frac{\partial T_{sr}}{\partial t} \tag{24}$$

$where$: λ_{srfm}: comprehensive thermal conductivity (surrounding space and formation)

These partial differential equations involved in the model, lead to a higher complexity to find a solution by using analytical models. Therefore, The mathematical model is discretized in space and time by applying the Crank-Nicholson finite differences method for two dimensions: (Ali et al., 2017)

a. *Fluid Inside Drill Pipe*

$$\begin{aligned}\frac{\rho_f c_f v_p}{2}\frac{\left[\left(T_{fp,j-1}^{n+1} - T_{fp,j}^{n+1}\right) + \left(T_{fp,j-1}^{n} - T_{fp,j}^{n}\right)\right]}{\Delta z} \\ + \frac{h_1\left[\left(T_{p,j}^{n+1} - T_{fp,j}^{n+1}\right) + \left(T_{p,j}^{n} - T_{fp,j}^{n}\right)\right]}{r_{pi}} + \frac{Q_p}{\pi r_{pi}^2} \\ = \rho_{fp} c_{fp} \frac{\left(T_{fp,j}^{n+1} - T_{fp,j}^{n}\right)}{\Delta t}\end{aligned} \tag{25}$$

b. Drill Pipe Wall

$$\frac{\lambda_p}{2\Delta z^2}\left[\left(T_{p,j+1}^{n+1}-2T_{p,j}^{n+1}+T_{p,j-1}^{n+1}\right)+\left(T_{p,j+1}^{n}-2T_{p,j}^{n}+T_{p,j-1}^{n}\right)\right]
+\frac{r_{po}h_{po}\left[\left(T_{a,j}^{n+1}-T_{p,j}^{n+1}\right)+\left(T_{a,j}^{n}-T_{p,j}^{n}\right)\right]}{\left(r_{po}^2-r_{pi}^2\right)}
+\frac{r_{pi}h_{pi}\left[\left(T_{fp,j}^{n+1}-T_{p,j}^{n+1}\right)+\left(T_{fp,j}^{n}-T_{p,j}^{n}\right)\right]}{\left(r_{po}^2-r_{pi}^2\right)}
=\rho_p c_p\frac{\left(T_{p,j}^{n+1}-T_{p,j}^{n}\right)}{\Delta t} \tag{26}$$

c. Fluid Inside Annular

$$\frac{\rho_l c_l v_a}{2}\frac{\left[\left(T_{a,j+1}^{n+1}-T_{a,j}^{n+1}\right)+\left(T_{a,j+1}^{n}-T_{a,j}^{n}\right)\right]}{\Delta z}
+\frac{r_{ci}h_{ci}\left[\left(T_{c,j}^{n+1}-T_{a,j}^{n+1}\right)+\left(T_{c,j}^{n}-T_{a,j}^{n}\right)\right]}{\left(r_{ci}^2-r_{po}^2\right)}
+\frac{r_{po}h_{po}\left[\left(T_{p,j}^{n+1}-T_{a,j}^{n+1}\right)+\left(T_{p,j}^{n}-T_{a,j}^{n}\right)\right]}{\left(r_{ci}^2-r_{po}^2\right)}
+\frac{Q_a}{\pi\left(r_{ci}^2-r_{po}^2\right)}=\rho_{fa}c_{fa}\frac{\left(T_{a,j}^{n+1}-T_{a,j}^{n}\right)}{\Delta t} \tag{27}$$

d. First Casing

$$\frac{\lambda_c}{2\Delta z^2}\left[\left(T_{c,j+1}^{n+1}-2T_{c,j}^{n+1}+T_{c,j-1}^{n+1}\right)+\left(T_{c,j+1}^{n}-2T_{c,j}^{n}+T_{c,j-1}^{n}\right)\right] + \frac{\lambda_{csr}\left[\left(T_{sr,j}^{n+1}-T_{c,j}^{n+1}\right)+\left(T_{sr,j}^{n}-T_{c,j}^{n}\right)\right]}{\left(r_{co}^2-r_{ci}^2\right)} + \frac{r_{ci}h_{ci}\left[\left(T_{a,j}^{n+1}-T_{c,j}^{n+1}\right)+\left(T_{a,j}^{n}-T_{c,j}^{n}\right)\right]}{\left(r_{co}^2-r_{ci}^2\right)} = \rho_c c_c\frac{\left(T_{c,j}^{n+1}-T_{c,j}^{n}\right)}{\Delta t} \qquad (28)$$

e. Surrounding Space

$$\frac{\lambda_{sr}}{2\Delta z^2}\left[\left(T_{sr,j+1}^{n+1}-2T_{sr,j}^{n+1}+T_{sr,j-1}^{n+1}\right)+\left(T_{sr,j+1}^{n}-2T_{sr,j}^{n}+T_{sr,j-1}^{n}\right)\right] + \frac{\lambda_{srfm}}{2r_{sr}\left(r_{sr}-r_{co}\right)}\left[\frac{\left(T_{c,j}^{n+1}-T_{sr,j}^{n+1}\right)+\left(T_{c,j}^{n}-T_{sr,j}^{n}\right)}{ln\left(\frac{r_{sr}}{r_{co}}\right)} - \frac{\left(T_{sr,j}^{n+1}-T_{fm,j}^{n+1}\right)+\left(T_{sr,j}^{n}-T_{fm,j}^{n}\right)}{ln\left(\frac{r_{fm}}{r_{sr}}\right)}\right] = \rho_{sr} c_{sr}\frac{\left(T_{sr,j}^{n+1}-T_{sr,j}^{n}\right)}{\Delta t} \qquad (29)$$

The code implemented was validated using Landmark to generate the temperature profile of a real well with operation parameters as follows:

Table 1. Parameters for drilling validation case

$fluid\ density\ (\rho_f): 1800\ \frac{kg}{m^3}$	$fluid\ inlet\ temperature\ (T_{in}): 26.672\ °C$
$water\ depth\ (wd): 145.9\ m$	$surface\ temperature\ (T_s): 25\ °C$
$flow\ rate\ (q): 900\ \frac{gal}{min}$	$geothermal\ gradient\ (TG_f): 0.024436\ °C/m$
$TG_w: -0.130226\ °C/m$	

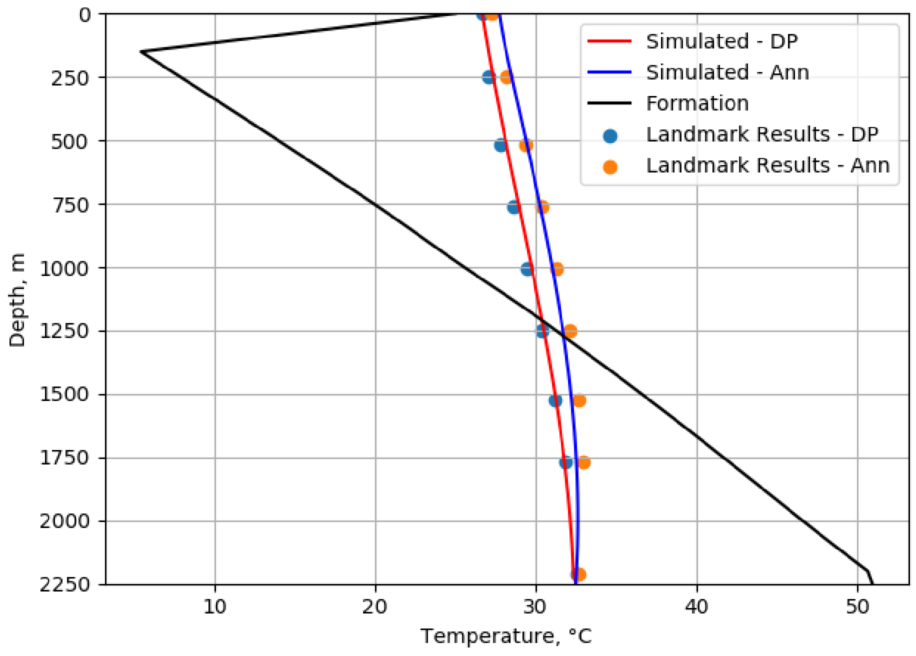


Figure 14. Validation of the drilling module

3.2.1. Heat Source Terms

During a drilling operation, heat is generated due to frictional pressure losses inside the drill string, the annular and drill bit. An empirical method was applied by (Keller et al., 1973) in order to estimate the amount of heat generated for each section as follows: 20% of the drilling pump hydraulic power losses in drill pipes, 8.5% losses in the annulus, and 70% losses at the drill bit.

(Marshall & Bentsen, 1982) derived the following formulas to calculate circulating pressure drops:

- *Drill pipe:*

$$\Delta P_p = \left(\frac{2f\rho v^2 L}{D g_c}\right) \tag{30}$$

$where$:

g_c: $127.094 \times 10^6 \frac{m \cdot s^2}{h^2}$

- *Annulus:*

$$\Delta P_a = \frac{2KL}{(r_{ci} - r_{po})g_c}\left[\frac{2(n+1)q}{n\pi(r_{ci} - r_{po})(r_{ci} - r_{po})^2}\right]^n \tag{31}$$

- *Drill bit:*

$$\Delta P_{bit} = \frac{\rho}{2g}\left(\frac{q/3600}{0.095 A_n}\right)^2 \tag{32}$$

Knowing that hydraulic power can be expressed as:

$$Power = \Delta P \cdot q \tag{33}$$

Besides (Kumar & Samuel, 2013) proposed an equation to estimate the heat generated in the drill string because of wellbore friction:

$$Q_p = 2\pi \cdot \omega \cdot M \tag{34}$$

In addition, (Nguyen et al., 2010) presented a formula to quantify the heat source generated by friction at the drill bit:

$$Q_{bit} = \frac{1}{J}(1 - \eta)\left(WOB \cdot \frac{ROP}{3600} + 2\pi\omega T_{bit}\right) \tag{35}$$

Finally integrating the heat sources shown above, we have:

- *Drill pipe:*

$$Q_p = 2\pi \cdot \omega \cdot M + 0.2 \cdot q \cdot \left(\frac{2f\rho v^2 L}{D g_c}\right) \tag{36}$$

- *Annulus:*

$$Q_a = 0.085 \frac{2KL}{(r_{ci} - r_{po})g_c}\left[\frac{2(n+1)q}{n\pi(r_{ci} - r_{po})(r_{ci} - r_{po})^2}\right]^n \tag{37}$$

- *Drill bit:*

$$Q_{bit} = \frac{1}{J}(1-\eta)\left(WOB \cdot \frac{ROP}{3600} + 2\pi\omega T_{bit}\right) + 0.7\frac{\rho}{2g}\left(\frac{q/3600}{0.095A_n}\right)^2 \tag{38}$$

3.2.2. Heat Transfer Coefficients

The Nusselt number (Nu) is used to determine the convective heat transfer coefficient. Moreover, Nu differs between drill string and in the annulus (Sieder & Tate, 1936).

$$h = \frac{\lambda}{2r} Nu \tag{39}$$

$where$:

h: convective heat transfer coefficient

λ: thermal conductivity

- *Calculation for Drill String:*

(Keller et al., 1973; Marshall & Bentsen, 1982) utilized a constant value when laminar regime presented.

$$Nu = 4.12\ if\ Re \leq 2300 \tag{40}$$

Nevertheless, (García et al., 1998; Santoyo-Gutierrez, 1997) employed a different value.

$$Nu = 4.36\ if\ Re \leq 2300 \tag{41}$$

(Petukhov, 1970) proposed a precise formula when the flow was fully turbulent ($Re > 10^4$). (Gnielinski, 1976) applied it later to a “transitional” flow regime.

$$Nu = \frac{(f/8)(Re - 1000)Pr}{1 + 12.7(f/8)^{\frac{1}{2}}\left(Pr^{\frac{2}{3}} - 1\right)} \; if \; 2300 < Re < 10^4 \tag{42}$$

$where$:

f: $friction \; factor$

Pr: $Prandtl \; number$

Re: $Reynolds \; number$

Reynolds and Prandtl numbers are defined as follows (Bergman et al., 2011):

$$Re = \frac{\rho v D}{\mu} \tag{43}$$

$$Pr = \frac{\mu c}{\lambda} \tag{44}$$

(Dittus & Boelter, 1985) introduced a simplified model in order to calculate Nu in turbulent flow. Then (Sieder & Tate, 1936) introduced the term $\left(\frac{\mu}{\mu_s}\right)^{0.14}$ to consider the variation of fluid properties.

$$Nu = 0{,}027 Re^{\frac{4}{5}} Pr^{\frac{1}{3}} \left(\frac{\mu}{\mu_s}\right)^{0.14} \; if \; Re \geq 10^4 \tag{45}$$

- *Calculation for Annulus:*

The heat transfer coefficient for laminar regime is calculated as follows (Yang et al., 2015):

$$Nu = 1.86(RePr)^{\frac{1}{3}} \left(\frac{D_H}{L}\right)^{\frac{1}{3}} \left(\frac{\mu}{\mu_w}\right)^{0.14} \; if \; Re < 2300 \tag{46}$$

Besides, Gnielinsky's correlation is applied to the conditions of transitional and turbulent flow.

3.2.3. Torque and Drag

A certain amount of heat is generated by friction between drill string and wellbore. The torque is involved in the calculation of this heat source, as shown in Eq. (36). Furthermore, it is very important to determine the friction factor, (Samuel, 2010) reported the range of values using different drilling fluids as shown in Table 2.

Table 2. Friction factors.

Fluid type	**Friction Factors**	
	Cased hole	**Open hole**
Oil-based	$0.16 - 0.20$	$0.17 - 0.25$
Water-based	$0.25 - 0.35$	$0.25 - 0.40$
Brine	$0.30 - 0.4$	$0.3 - 0.4$
Polymer-based	$0.15 - 0.22$	$0.2 - 0.3$
Synthetic-based	$0.12 - 0.18$	$0.15 - 0.25$
Foam	$0.30 - 0.4$	$0.35 - 0.55$
Air	$0.35 - 0.55$	$0.40 - 0.60$

Besides, after analyzing 33 wells in a Norwegian platform, (Craig, 2003) determined that μ = 0.24 was the most applicable friction factor for the different types of wells, and there was an insignificant impact due to the condition of the well trajectory.

On the other hand, a buoyancy effect acts directly on the generated forces. (Aadnoy et al., 2010) defined the buoyancy factor (B_y) when the drilling fluid density in the drill string is not the same to the fluid density in the annular. This factor will affect the effective weight of the pipe.

$$B_y = 1 - \frac{\rho_o A_o - \rho_i A_i}{\rho_{pipe}(A_o - A_i)} \tag{47}$$

$$W_{eff} = B_y W_d \tag{48}$$

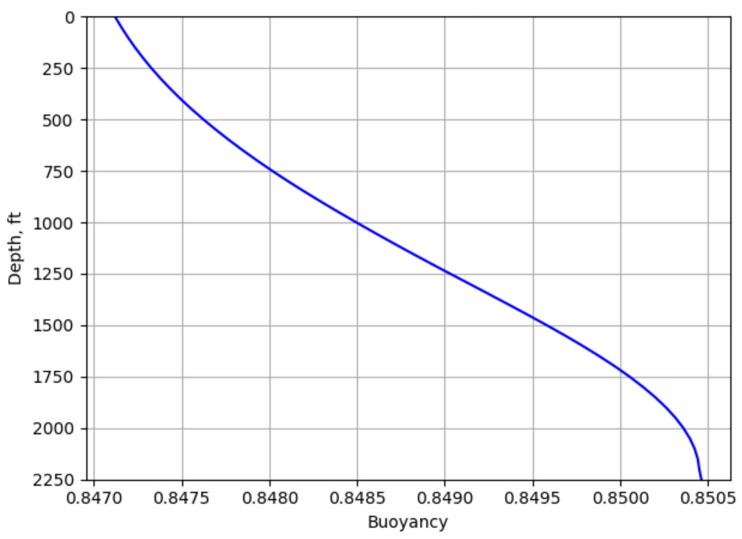


Figure 15. Buoyancy factor profile

A very well-known torque and drag model (Johancsik et al., 1984) is considered within this work. It is defined as follows:

$$F_n = \left[(F * \Delta\alpha * sin\bar{\theta})^2 + (F * \Delta\bar{\theta} + W_{eff} * sin\bar{\theta})^2\right]^{\frac{1}{2}} \tag{49}$$

- *Drag force:*

$$\Delta F_t = W_{eff} * cos\bar{\theta} \pm \mu_a * F_n \tag{50}$$

Where μ_a is the axial coefficient of friction and it is given as follows:

$$\mu_a = \mu_o \frac{v_a}{\sqrt{(\omega r)^2 + v_a^2}} \tag{51}$$

$where$:	μ_o: coefficient of friction between $formation, casing\ and\ string$	ω: angular frequency
	r: $outer\ radius\ of\ the\ tubular$	v_a: $axial\ speed$

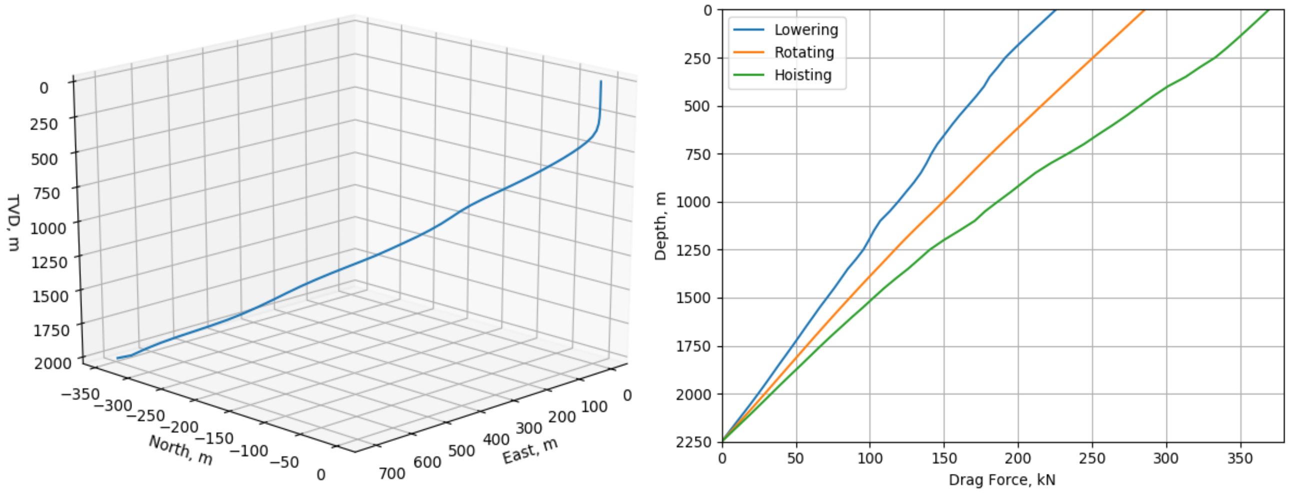


Figure 16. Drag Forces along the well

- *Torque:*

$$\Delta M = \mu_t * F_n * r \tag{52}$$

Where μ_t is the tangential coefficient of friction and it is given as follows:

$$\mu_t = \mu_o \frac{\omega . r}{\sqrt{(\omega r)^2 + v_a^2}} \tag{53}$$

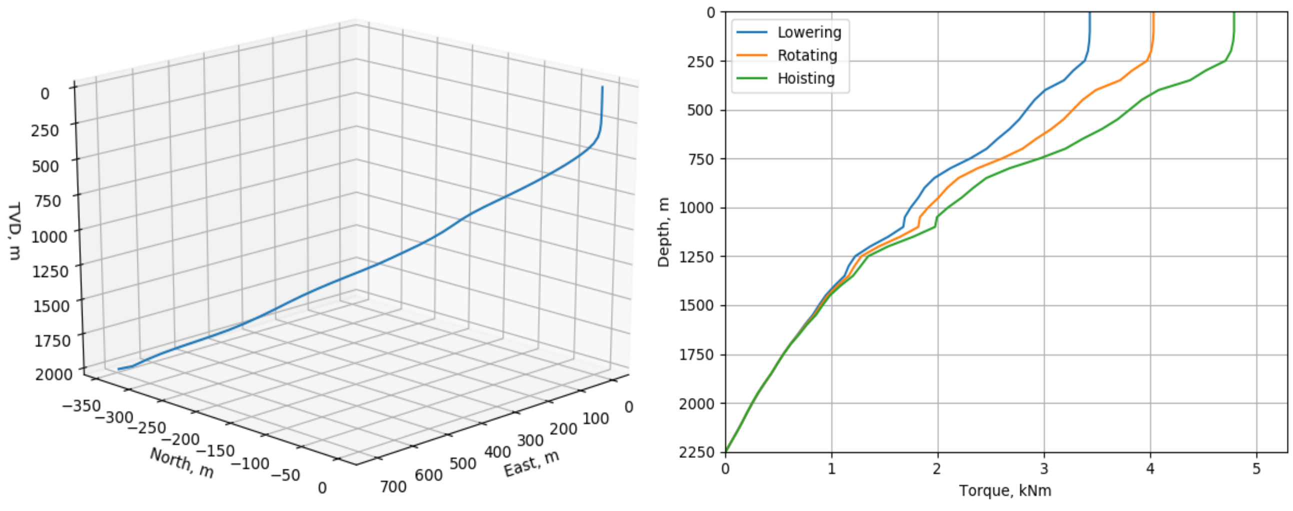


Figure 17. Torque along the well

3.2.4. Viscosity

Drilling fluids suffer changes in viscosity depending on pressure and temperature conditions. It is also important to consider the difference in the heat transfer process between viscosity behavior for Newtonian and non-Newtonian fluids (Santoyo et al., 2003).

$$\tau = \tau_0 + K\gamma^n \tag{54}$$

where:

τ: shear stress	K: consistency index
τ_0: yield stress	γ: shear rate
n: flow index	

Pressure and temperature conditions will affect the values of the coefficients τ_0, K and n. Hence, their values are determined at laboratory. Once these values are determined, the apparent viscosity can be calculated as follows:

$$\mu_{app} = \tau_0\gamma^{-1} + K\gamma^{n-1} \tag{55}$$

The shear rate value is obtained from Eq. (54) and shear stress is required. An improved model for Herschel-Bulkley fluids was developed (Fan et al., 2014). However, the traditional model to determine the wall shear stress (τ_w) was considered for faster and more convenient integration to the package.

$$q_p = \frac{n\pi r^3}{3n+1}\left(\frac{\tau_w}{K}\right)^{\frac{1}{n}}\left[1 - \frac{3n+1}{n(2n+1)}\frac{\tau_0}{\tau_w}\right] \tag{56}$$

$$q_a = \frac{n\pi {r_\delta}^2}{2(2n+1)K^{\frac{1}{n}}}(r_o + r_i)\left(\tau_w - \frac{2n+1}{n+1}\tau_0\right)^{\frac{1}{n}} \tag{57}$$

3.2.5. Effects of the rotary drill pipe with its eccentricity on the pressure drop in the annulus

The effective forces along directional wells cause the variability in its eccentricity. Vertical sections would present a value of $e = 0$, while build-up segments should have eccentricities between the range 0 and 1, and horizontal sectors are totally eccentric ($e = 1$). This property will affect the pressure drop behavior, this work implemented a ratio of the pressure drop in the annulus with a rotary inner pipe and stationary inner pipe (Cartalos & Dupuis, 1993).

$$R_{rot} = \frac{(dP/dL)_{rot}}{(dP/dL)_{non-rot}} = \left[1 + \frac{3}{2}\left(\frac{S_o(z)}{d}\right)^2\right]^{1/2} \tag{58}$$

$where$:

$S_o(z)$: amplitude of inner pipe at depth z $\qquad$ $d = r_o - r_i$

This amplitude is replaceable with an average value, so the mean eccentricity can be applied to the entire well.

$$e_{avg} = S_{avg}/d \tag{59}$$

Moreover, two correction coefficients were introduced to consider the effects of rotation and eccentricity on the pressure drop due to fluid flow through the annulus (M. Haciislamoglu & Langlinais, 1990).

$$R_{ecc} = \frac{(dP/dL)_{ecc}}{(dP/dL)_{con}} \tag{60}$$

- *Laminar regime:*

$$R_{ecc} = 1 - 0.072\frac{e}{n}\left(\frac{D_i}{D_o}\right)^{0.8454} - 1.5e^2\sqrt{n}\left(\frac{D_i}{D_o}\right)^{0.1852} + 0.96e^3\sqrt{n}\left(\frac{D_i}{D_o}\right)^{0.2527} \quad (61)$$

In addition, the correction coefficients were modified when the flow is acting within a turbulent regime (Mustafa Haciislamoglu, 1994).

- *Turbulent regime:*

$$R_{ecc} = 1 - 0.048\frac{e}{n}\left(\frac{D_i}{D_o}\right)^{0.8454} - \frac{2}{3}e^2\sqrt{n}\left(\frac{D_i}{D_o}\right)^{0.1852} + 0.285e^3\sqrt{n}\left(\frac{D_i}{D_o}\right)^{0.2527} \quad (62)$$

3.3. Production

During production, produced fluids circulate from the bottom to the surface, which suppose a forced convection occurring inside the production tubing. However, completion fluid is placed in the annulus and it remains static; this where natural free convection shows up. Furthermore, heat conduction is considered along the radial axis (tubing, production fluid, casings, cement and formation) similarly as in the drilling case.

a. Fluid Inside Tubing

During the fluid extraction the elements suffer energy changes due to heat transfer within the production fluid because of the flow upwards in the axial direction, forced heat convection between the fluid and the inner wall of the tubing.

$$\rho_f c_f v_t \frac{\partial T_{ft}}{\partial z} + \frac{2h_1\left(T_t - T_{ft}\right)}{r_{ti}} + \frac{Q_t}{\pi r_{ti}^2} = \rho_{ft} c_{ft} \frac{\partial T_{ft}}{\partial t} \quad (63)$$

b. Tubing Wall

The heat balance along the tubing wall involves heat conduction in the axial direction, and heat convection between the inner wall of the production tubing and the fluid inside.

$$\frac{\partial}{\partial z}\left(\lambda_t \frac{\partial T_t}{\partial z}\right) + \frac{2r_{to}h_{to}(T_a - T_t)}{(r_{to}^2 - r_{ti}^2)} + \frac{2r_{ti}h_{ti}\left(T_{ft} - T_t\right)}{(r_{to}^2 - r_{ti}^2)} = \rho_t c_t \frac{\partial T_t}{\partial t} \tag{64}$$

c. Fluid Inside Annular

The heat balance in the annular section involves natural heat convection due to the static completion fluid, heat convection between the fluid and first casing or wellbore face, and heat convection between the fluid and the outer wall of the production tubing.

$$\frac{\partial}{\partial z}\left(\lambda_a \frac{\partial T_a}{\partial z}\right) + \frac{2r_{ci}h_{ci}(T_c - T_a)}{(r_{to}^2 - r_{ti}^2)} + \frac{2r_{to}h_{to}(T_t - T_a)}{(r_{to}^2 - r_{ti}^2)} = \rho_{fa} c_{fa} \frac{\partial T_a}{\partial t} \tag{65}$$

d. First Casing

For this case, the balance equation involves heat conduction in the axial direction, heat transfer with the first layer of cement sheath and heat convection with the completion fluid in the annulus.

$$\frac{\partial}{\partial z}\left(\lambda_c \frac{\partial T_c}{\partial z}\right) + \frac{2\lambda_{csr}(T_{sr} - T_c)}{(r_{co}^2 - r_{ci}^2)} + \frac{2r_{ci}h_{ci}(T_a - T_c)}{(r_{co}^2 - r_{ci}^2)} = \rho_c c_c \frac{\partial T_c}{\partial t} \tag{66}$$

e. Surrounding Space

The heat balance in this comprehensive section involves heat conduction in both axial and radial directions.

$$\frac{\partial}{\partial z}\left(\lambda_{sr} \frac{\partial T_{sr}}{\partial z}\right) + \frac{\partial}{\partial r}\left(\lambda_{srfm} r \frac{\partial T}{\partial r}\right) = \rho_{sr} c_{sr} \frac{\partial T_{sr}}{\partial t} \tag{67}$$

These partial differential equations involved in the model, lead to a higher complexity to find a solution by using analytical models. Therefore, the mathematical model is discretized in space and time by applying the Crank-Nicholson finite differences method for two dimensions (Ali et al., 2017):

a. Fluid Inside Tubing

$$\frac{\rho_f c_f v_t}{2}\frac{\left[\left(T_{ft,j+1}^{n+1}-T_{ft,j}^{n+1}\right)+\left(T_{ft,j+1}^{n}-T_{ft,j}^{n}\right)\right]}{\Delta z} + \frac{h_1\left[\left(T_{t,j}^{n+1}-T_{ft,j}^{n+1}\right)+\left(T_{t,j}^{n}-T_{ft,j}^{n}\right)\right]}{r_{ti}} + \frac{Q_t}{\pi r_{ti}^2} = \rho_{ft} c_{ft}\frac{\left(T_{ft,j}^{n+1}-T_{ft,j}^{n}\right)}{\Delta t} \tag{68}$$

b. Tubing Wall

$$\frac{\lambda_t}{2\Delta z^2}\left[\left(T_{t,j+1}^{n+1}-2T_{t,j}^{n+1}+T_{t,j-1}^{n+1}\right)+\left(T_{t,j+1}^{n}-2T_{t,j}^{n}+T_{t,j-1}^{n}\right)\right] + \frac{r_{to}h_{to}\left[\left(T_{a,j}^{n+1}-T_{t,j}^{n+1}\right)+\left(T_{a,j}^{n}-T_{t,j}^{n}\right)\right]}{\left(r_{to}^2-r_{ti}^2\right)} + \frac{r_{ti}h_{ti}\left[\left(T_{ft,j}^{n+1}-T_{t,j}^{n+1}\right)+\left(T_{ft,j}^{n}-T_{t,j}^{n}\right)\right]}{\left(r_{to}^2-r_{ti}^2\right)} = \rho_t c_t\frac{\left(T_{t,j}^{n+1}-T_{t,j}^{n}\right)}{\Delta t} \tag{69}$$

c. Fluid Inside Annular

$$\frac{\lambda_a}{2\Delta z^2}\left[\left(T_{a,j+1}^{n+1}-2T_{a,j}^{n+1}+T_{a,j-1}^{n+1}\right)+\left(T_{a,j+1}^{n}-2T_{a,j}^{n}+T_{a,j-1}^{n}\right)\right] + \frac{r_{ci}h_{ci}\left[\left(T_{c,j}^{n+1}-T_{a,j}^{n+1}\right)\left(T_{c,j}^{n}-T_{a,j}^{n}\right)\right]}{\left(r_{to}^2-r_{ti}^2\right)} + \frac{r_{to}h_{to}\left[\left(T_{t,j}^{n+1}-T_{a,j}^{n+1}\right)+\left(T_{t,j}^{n}-T_{a,j}^{n}\right)\right]}{\left(r_{to}^2-r_{ti}^2\right)} = \rho_{fa} c_{fa}\frac{\left(T_{t,j}^{n+1}-T_{t,j}^{n}\right)}{\Delta t} \tag{70}$$

d. First Casing

$$\frac{\lambda_c}{2\Delta z^2}\left[\left(T_{c,j+1}^{n+1} - 2T_{c,j}^{n+1} + T_{c,j-1}^{n+1}\right) + \left(T_{c,j+1}^{n} - 2T_{c,j}^{n} + T_{c,j-1}^{n}\right)\right] + \frac{\lambda_{csr}\left[\left(T_{sr,j}^{n+1} - T_{c,j}^{n+1}\right) + \left(T_{sr,j}^{n} - T_{c,j}^{n}\right)\right]}{\left(r_{co}^2 - r_{ci}^2\right)} + \frac{r_{ci}h_{ci}\left(T_{a,j}^{n+1} - T_{c,j}^{n+1}\right) + \left(T_{a,j}^{n} - T_{c,j}^{n}\right)}{\left(r_{co}^2 - r_{ci}^2\right)} = \rho_c c_c \frac{\left(T_{c,j}^{n+1} - T_{c,j}^{n}\right)}{\Delta t} \qquad (71)$$

e. Surrounding Space

$$\frac{\lambda_{sr}}{2\Delta z^2}\left[\left(T_{sr,j+1}^{n+1} - 2T_{sr,j}^{n+1} + T_{sr,j-1}^{n+1}\right) + \left(T_{sr,j+1}^{n} - 2T_{sr,j}^{n} + T_{sr,j-1}^{n}\right)\right] + \frac{\lambda_{srfm}}{2r_{sr}\left(r_{sr} - r_{co}\right)}\left[\frac{\left(T_{c,j}^{n+1} - T_{sr,j}^{n+1}\right) + \left(T_{c,j}^{n} - T_{sr,j}^{n}\right)}{ln\left(\frac{r_{sr}}{r_{co}}\right)} - \frac{\left(T_{sr,j}^{n+1} - T_{fm,j}^{n+1}\right) + \left(T_{sr,j}^{n} - T_{fm,j}^{n}\right)}{ln\left(\frac{r_{fm}}{r_{sr}}\right)}\right] = \rho_{sr} c_{sr} \frac{\left(T_{sr,j}^{n+1} - T_{sr,j}^{n}\right)}{\Delta t} \qquad (72)$$

The Production module was validated with data of a natural flowing well from the Permian Basin, West Texas (Sagar et al., 1991). Figure 18 shows the results. Operation parameters as follows:

$q_l \approx 600\frac{bbl}{d}$	$API: 34.3°$
$d_t: 2.875"$	$T_s: 24.4°C$
$TG_f: 0.01100491°C/m$	

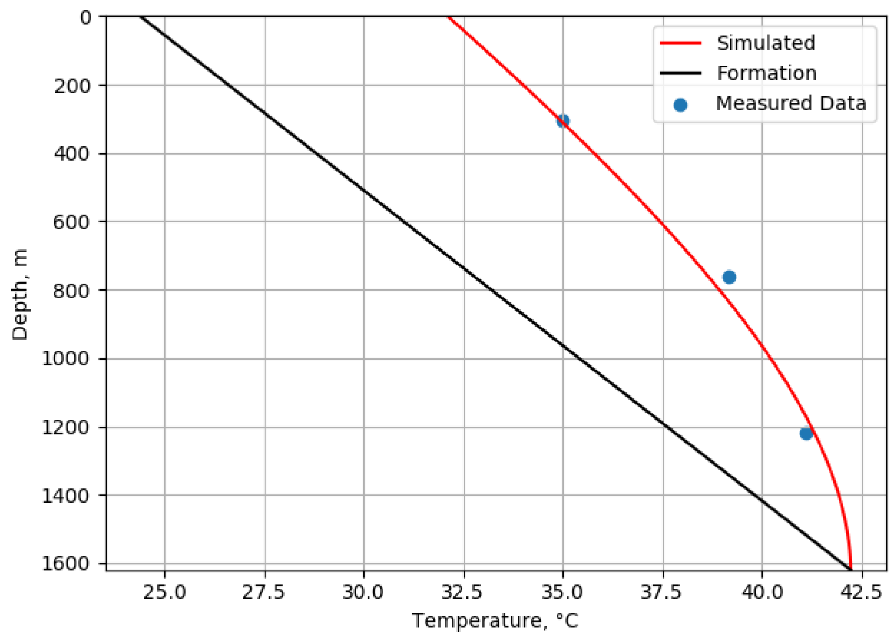


Figure 18. Validation of the production module

3.3.1. Heat Source Terms

During production, heat is generated due to frictional pressure losses inside the tubing. The calculation of this term is done by the same way as in the Drilling module, considering that only the circulation of fluid inside the tubing is acting this time (Marshall & Bentsen, 1982).

$$Q_t = 0.2 \cdot q \cdot \left(\frac{2f\rho v^2 L}{D g_c} \right) \tag{73}$$

3.3.2. Heat Transfer Coefficients

Heat transfer by forced convection takes place inside the production tubing, where produced fluids are circulating upwards to the surface. This condition is mainly influenced by the effect of the Reynolds number (Re). On the other hand, the annular section has completion fluid without flowing, natural convection takes place here and it will be affected by the effect of the Grashof number (Gr). This parameter is defined as the relation between buoyancy and viscous forces in the velocity boundary layer, and it is given as (Bergman et al., 2011):

$$Gr = \frac{g\alpha(T_s - T_\infty)\rho^2 L^3}{\mu^2} \tag{74}$$

Besides, the Rayleigh number (Ra) is also utilized when free convection happens. Heat transfer through fluids can take place by conduction and convection, the Ra express which one is dominating the process if a temperature gradient is there. The Rayleigh number is given as (Bergman et al., 2011):

$$Ra = GrPr \tag{75}$$

A forced convection correlation (Gnielinski, 1976) is implemented in this work for calculating the Nusselt number involved in the fluid circulation inside the production tubing, for transitional and fully turbulent flow regimes:

$$Nu = \frac{(f/8)(Re - 1000)Pr}{1 + 12.7(f/8)^{\frac{1}{2}}\left(Pr^{\frac{2}{3}} - 1\right)} \; if\ turbulent\ flow\ (Re > 2300) \tag{76}$$

The value of Nu for laminar conditions is often utilized as a constant (García et al., 1998; Santoyo-Gutierrez, 1997):

$$Nu = 4.36\ if\ laminar\ flow\ (Re < 2300) \tag{77}$$

Regarding the annular space, a correlation was developed for natural convection considering the inclination angles (Dropkin & Somerscales, 1965):

$$Nu = C(Ra)^{\frac{1}{3}}Pr^{0.074} \tag{78}$$

Where C is the coefficient of the inclination and it is given as shown in Table 3. For any section with an inclination between the values presented in the table, the algorithm interpolates the points and retrieves a more adequate coefficient.

Table 3. Free/natural convection correlations for concentric annulus for various inclinations

C	Inclination θ
$\mathbf{0.069}$	0
$\mathbf{0.065}$	30
$\mathbf{0.059}$	45
$\mathbf{0.057}$	60
$\mathbf{0.049}$	90

3.4. Injection

During injection, fluids are circulated from surface to the bottom, which suppose a forced convection occurring inside the tubing. However, completion fluid is placed in the annulus and it remains static; this where natural free convection shows up. Furthermore, heat conduction is considered along the radial axis (tubing, production fluid, casings, cement and formation) similarly as in the drilling and production cases.

a. Fluid Inside Tubing

During the fluid injection, all the segments suffer energy changes due to heat transfer within the injected fluid because of the flow downwards in the axial direction, forced heat convection between the fluid and the inner wall of the tubing, transporting heat at a much faster rate than conduction (Wooley, 1980).

$$-\rho_f c_f v_t \frac{\partial T_{ft}}{\partial z} + \frac{2h_1\left(T_t - T_{ft}\right)}{r_{pi}} + \frac{Q_p}{\pi r_{pi}^2} = \rho_l c_l \frac{\partial T_{ft}}{\partial t} \tag{79}$$

b. Tubing Wall

The heat balance along the tubing wall involves heat conduction in the axial direction, and heat convection between the inner wall of the injection tubing and the fluid inside.

$$\frac{\partial}{\partial z}\left(\lambda_t \frac{\partial T_t}{\partial z}\right) + \frac{2r_{to}h_{to}(T_a - T_t)}{(r_{to}^2 - r_{ti}^2)} + \frac{2r_{ti}h_{ti}\left(T_{ft} - T_t\right)}{(r_{to}^2 - r_{ti}^2)} = \rho_t c_t \frac{\partial T_t}{\partial t} \tag{80}$$

c. Fluid Inside Annular

The heat balance in the annular section involves natural heat convection due to the static completion fluid, heat convection between the fluid and first casing or wellbore face, and heat convection between the fluid and the outer wall of the injection tubing.

$$\frac{\partial}{\partial z}\left(\lambda_a \frac{\partial T_a}{\partial z}\right) + \frac{2r_{ci}h_{ci}(T_c - T_a)}{(r_{to}^2 - r_{ti}^2)} + \frac{2r_{to}h_{to}(T_t - T_a)}{(r_{to}^2 - r_{ti}^2)} = \rho_{fa} c_{fa} \frac{\partial T_a}{\partial z} \tag{81}$$

d. First Casing

In this case, the balance equation involves heat conduction in the axial direction, heat transfer with the first layer of cement sheath and heat convection with the completion fluid in the annulus.

$$\frac{\partial}{\partial z}\left(\lambda_c \frac{\partial T_c}{\partial z}\right) + \frac{2\lambda_{csr}(T_{sr} - T_c)}{(r_{co}^2 - r_{ci}^2)} + \frac{2r_{ci}h_{ci}(T_a - T_c)}{(r_{co}^2 - r_{ci}^2)} = \rho_c c_c \frac{\partial T_c}{\partial t} \tag{82}$$

e. Surrounding Space

The heat balance in this comprehensive section involves heat conduction in both axial and radial directions (Zhang et al., 2018).

$$\frac{\partial}{\partial z}\left(\lambda_{sr} \frac{\partial T_{sr}}{\partial z}\right) + \frac{\partial}{\partial r}\left(\lambda_{srfm} r \frac{\partial T}{\partial r}\right) = \rho_{sr} c_{sr} \frac{\partial T_{sr}}{\partial t} \tag{83}$$

These partial differential equations involved in the model, lead to a higher complexity to find a solution by using analytical models. Therefore, the mathematical model is discretized in space and time by applying the Crank-Nicholson finite differences method for two dimensions (Ali et al., 2017):

a. Fluid Inside Tubing

$$\begin{aligned} &\frac{\rho_f c_f v_t}{2} \frac{\left[\left(T_{ft,j-1}^{n+1} - T_{ft,j}^{n+1}\right) + \left(T_{ft,j-1}^{n} - T_{ft,j}^{n}\right)\right]}{\Delta z} \\ &+ \frac{h_1\left[\left(T_{p,j}^{n+1} - T_{fp,j}^{n+1}\right) + \left(T_{p,j}^{n} - T_{fp,j}^{n}\right)\right]}{r_{pi}} + \frac{Q_p}{\pi r_{pi}^2} \\ &= \rho_l c_l \frac{\left(T_{ft,j}^{n+1} - T_{ft,j}^{n}\right)}{\Delta t} \end{aligned} \tag{84}$$

b. Tubing Wall

$$\frac{\lambda_t}{2\Delta z^2}\left[\left(T_{t,j+1}^{n+1}-2T_{t,j}^{n+1}+T_{t,j-1}^{n+1}\right)+\left(T_{t,j+1}^{n}-2T_{t,j}^{n}+T_{t,j-1}^{n}\right)\right] + \frac{r_{to}h_{to}\left[\left(T_{a,j}^{n+1}-T_{t,j}^{n+1}\right)+\left(T_{a,j}^{n}-T_{t,j}^{n}\right)\right]}{\left(r_{to}^2-r_{ti}^2\right)} + \frac{r_{ti}h_{ti}\left[\left(T_{ft,j}^{n+1}-T_{t,j}^{n+1}\right)+\left(T_{ft,j}^{n}-T_{t,j}^{n}\right)\right]}{\left(r_{to}^2-r_{ti}^2\right)} = \rho_t c_t \frac{\left(T_{t,j}^{n+1}-T_{t,j}^{n}\right)}{\Delta t} \tag{85}$$

c. Fluid Inside Annular

$$\frac{\lambda_a}{2\Delta z^2}\left[\left(T_{a,j+1}^{n+1}-2T_{a,j}^{n+1}+T_{a,j-1}^{n+1}\right)+\left(T_{a,j+1}^{n}-2T_{a,j}^{n}+T_{a,j-1}^{n}\right)\right] + \frac{r_{ci}h_{ci}\left[\left(T_{c,j}^{n+1}-T_{a,j}^{n+1}\right)\left(T_{c,j}^{n}-T_{a,j}^{n}\right)\right]}{\left(r_{to}^2-r_{ti}^2\right)} + \frac{r_{to}h_{to}\left[\left(T_{t,j}^{n+1}-T_{a,j}^{n+1}\right)+\left(T_{t,j}^{n}-T_{a,j}^{n}\right)\right]}{\left(r_{to}^2-r_{ti}^2\right)} = \rho_{fa} c_{fa} \frac{\left(T_{t,j}^{n+1}-T_{t,j}^{n}\right)}{\Delta t} \tag{86}$$

d. First Casing

$$\frac{\lambda_c}{2\Delta z^2}\left[\left(T_{c,j+1}^{n+1}-2T_{c,j}^{n+1}+T_{c,j-1}^{n+1}\right)+\left(T_{c,j+1}^{n}-2T_{c,j}^{n}+T_{c,j-1}^{n}\right)\right] + \frac{\lambda_{csr}\left[\left(T_{sr,j}^{n+1}-T_{c,j}^{n+1}\right)+\left(T_{sr,j}^{n}-T_{c,j}^{n}\right)\right]}{\left(r_{co}^2-r_{ci}^2\right)} + \frac{r_{ci}h_{ci}\left(T_{a,j}^{n+1}-T_{c,j}^{n+1}\right)+\left(T_{a,j}^{n}-T_{c,j}^{n}\right)}{\left(r_{co}^2-r_{ci}^2\right)} = \rho_c c_c \frac{\left(T_{c,j}^{n+1}-T_{c,j}^{n}\right)}{\Delta t} \tag{87}$$

e. Surrounding Space

$$\frac{\lambda_{sr}}{2\Delta z^2}\left[\left(T_{sr,j+1}^{n+1}-2T_{sr,j}^{n+1}+T_{sr,j-1}^{n+1}\right)+\left(T_{sr,j+1}^{n}-2T_{sr,j}^{n}+T_{sr,j-1}^{n}\right)\right]$$
$$+\frac{\lambda_{srfm}}{2r_{sr}(r_{sr}-r_{co})}\left[\frac{\left(T_{c,j}^{n+1}-T_{sr,j}^{n+1}\right)+\left(T_{c,j}^{n}-T_{sr,j}^{n}\right)}{ln\left(\frac{r_{sr}}{r_{co}}\right)}\right. \tag{88}$$
$$\left.-\frac{\left(T_{sr,j}^{n+1}-T_{fm,j}^{n+1}\right)+\left(T_{sr,j}^{n}-T_{fm,j}^{n}\right)}{ln\left(\frac{r_{fm}}{r_{sr}}\right)}\right]$$
$$=\rho_{sr}c_{sr}\frac{\left(T_{sr,j}^{n+1}-T_{sr,j}^{n}\right)}{\Delta t}$$

The injection module developed in this work was validated with data of a water injection well provided by Nowak (Nowak, 1953). Figure 19 shows the comparisons between the simulation and the measured data, where the model captures the measurements very well. Operation parameters as follows:

q_l: $900\frac{bbl}{d}$	T_s: $20°C$
T_{in}: $28.1°C$	TG_f: $0.036°C/m$

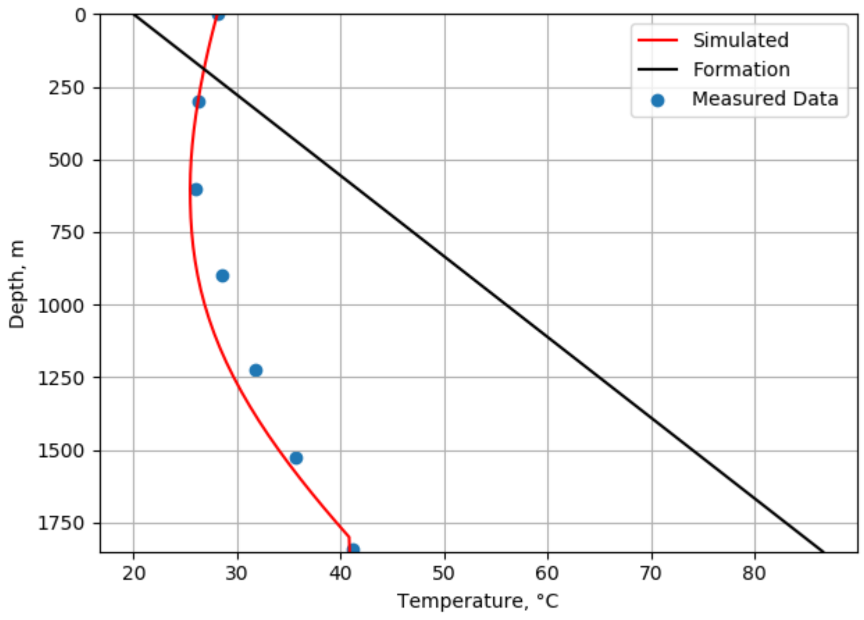


Figure 19. Validation of the injection module

3.4.1. Heat Source Terms

During injection, heat is generated due to frictional pressure losses inside the tubing. The calculation of this term is done by the same way as in drilling and production modules, considering that only the circulation of fluid inside the tubing is acting (Eq. (73)).

3.4.2. Heat Transfer Coefficients

The injection operation involves a pretty similar behavior as during production regarding heat transfer, fluid circulation inside the tubing generates forced convection while static completion fluid in the annular space implies natural or free convection. Therefore, the same equations for coefficients calculations are applied.

4. Simulation Sensitivity Analysis

A sensitivity study is carried out for the three well operations presented in this thesis. Initially, a base case is defined, and main parameters (flow rate, specific heat capacity, thermal conductivity, density and viscosity) are individually evaluated and discussed.

The following base wellbore profile (wellpath) is defined to run the analysis:

- Type: "J"
- Target depth: $3000\ m$
- Kick-off point (KOP): $1200\ m$
- End of build (EOB): $1500\ m$
- Build angle: $45°$
- No change in Azimuth

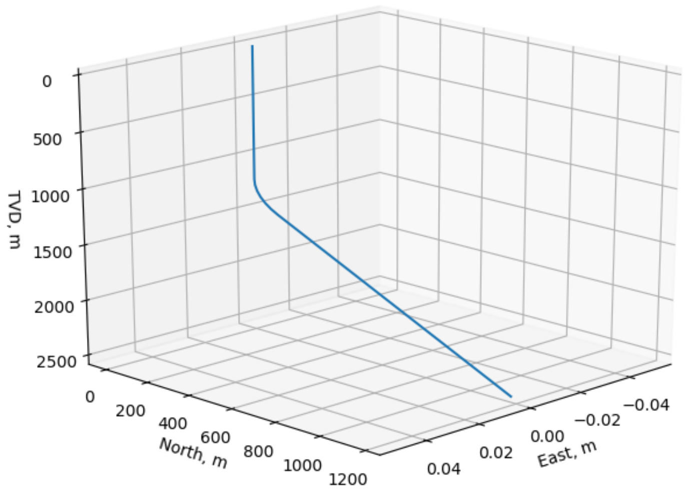


Figure 20. Base well profile to run sensitivity analysis

The objective is to estimate the impact of the analyzed parameters on the temperature distribution along the well. This analysis is performed through a comparison between a defined base case and modified cases for the respective parameter.

4.1. Drilling

4.1.1. Base case

For the drilling base case, the surface temperature is set at $25\ °C$ and the temperature of the fluid at the drill pipe inlet is $20\ °C$. The simulations are performed for $10\ hours$ of operation with a circulation rate of $1000\ \frac{l}{min}$, mud specific heat capacity of $4000\ \frac{J}{kg\cdot °C}$, mud thermal conductivity of $0.635\ \frac{W}{m\cdot °C}$ and fluid density of $1198\ \frac{kg}{m^3}$.

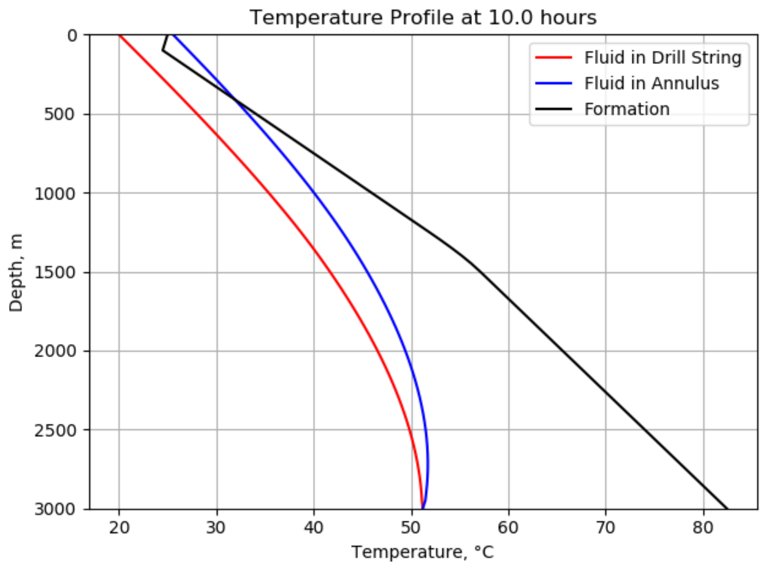


Figure 21. Drilling base case – temperature profile

4.1.2. Flow rate

The mud circulation rate is certainly one of the most important parameters affecting the heat transfer processes. It can drastically change frictional pressure losses along the pipe and thus affect pressure dependent properties such as density and viscosity. The flow rate is analyzed by implementing a variation from $500\ \frac{l}{min}\ to\ 3000\ \frac{l}{min}$.

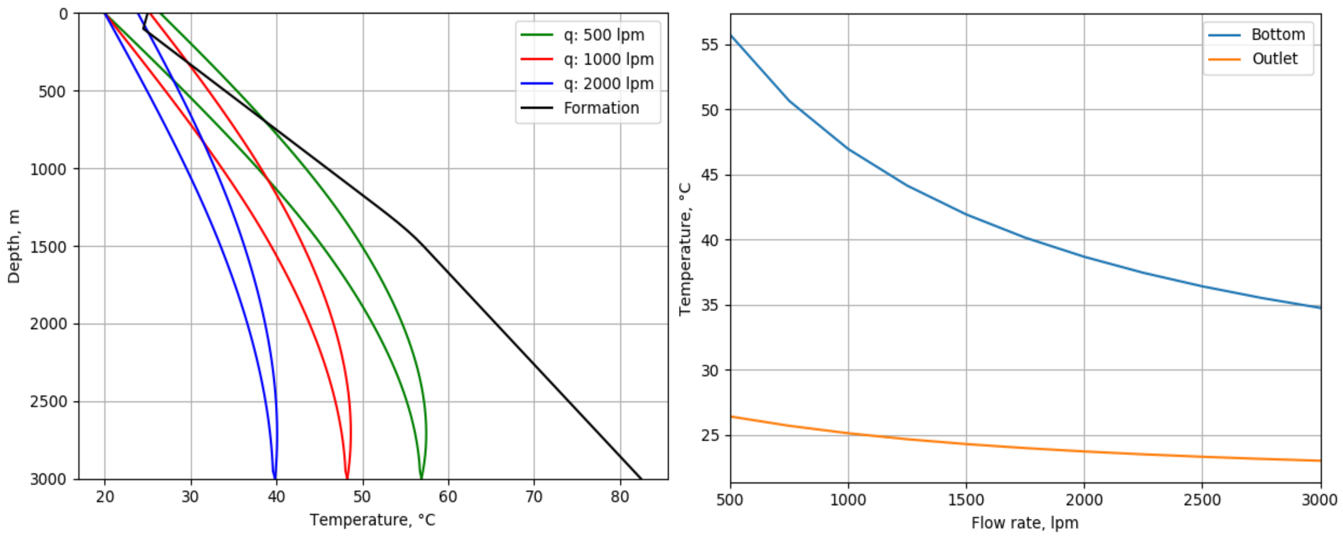


Figure 22. Flow rate effect on temperature profile during drilling

The results confirm the significant effect of this parameter with the highest impact on the bottom hole temperature. The increases to $1000\ \frac{l}{min}$ and $2000\ \frac{l}{min}$ produce reductions at bottom of around $8\ °C$ and $17\ °C$, respectively. The bottom hole temperature decreases around 36% due to the variation from the base case up to a flow rate of $3000\ \frac{l}{min}$, while the outlet temperature at the annulus present a small change of $4\ °C$ approximately. Therefore, the increase of the circulation rate causes a significant reduction on the well temperature profile.

4.1.3. Specific heat capacity

Specific heat capacity of the drilling mud refers to the amount of heat required per unit of mass to increase its temperature by one degree. The analysis in performed variating the specific heat capacity within a range from $2000\ \frac{J}{kg \cdot °C}$ to $4500\ \frac{J}{kg \cdot °C}$.

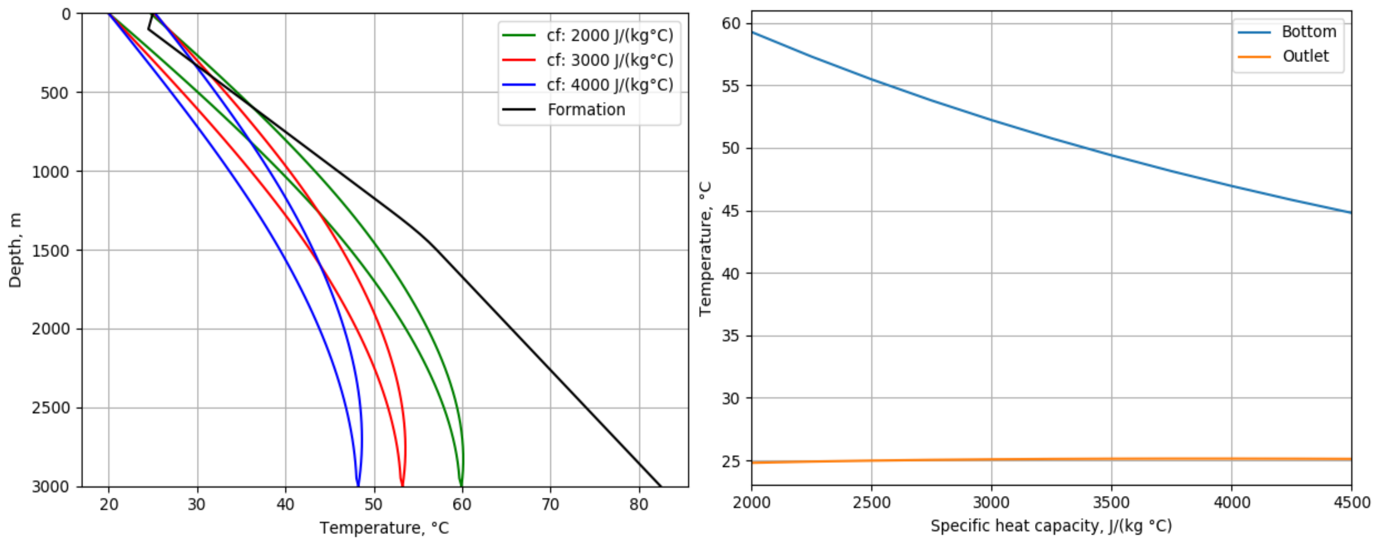


Figure 23. Specific heat capacity effect on temperature profile during drilling

As displayed in Figure 23, the results show a considerable increase in the well temperature profile when the specific heat capacity decreases. It makes sense, since a lower value of heat capacity indicates than less heat is required to increment de fluid temperature. An increment of almost $15\,°C$ is produced due the specific heat capacity reducing from $4500\ \frac{J}{kg\cdot°C}$ to $2000\ \frac{J}{kg\cdot°C}$, a change of 25%. On the other hand, the outlet temperature at surface remains almost constant.

4.1.4. Thermal conductivity

Thermal conductivity of a material is defined as the ability to conduct heat. In this case, the thermal conductivity defines how easily the heat transfer can proceed through the drilling mud. The analysis in performed variating this parameter within a range from $0.2\ \frac{W}{m\cdot°C}$ to $1.0\ \frac{W}{m\cdot°C}$.

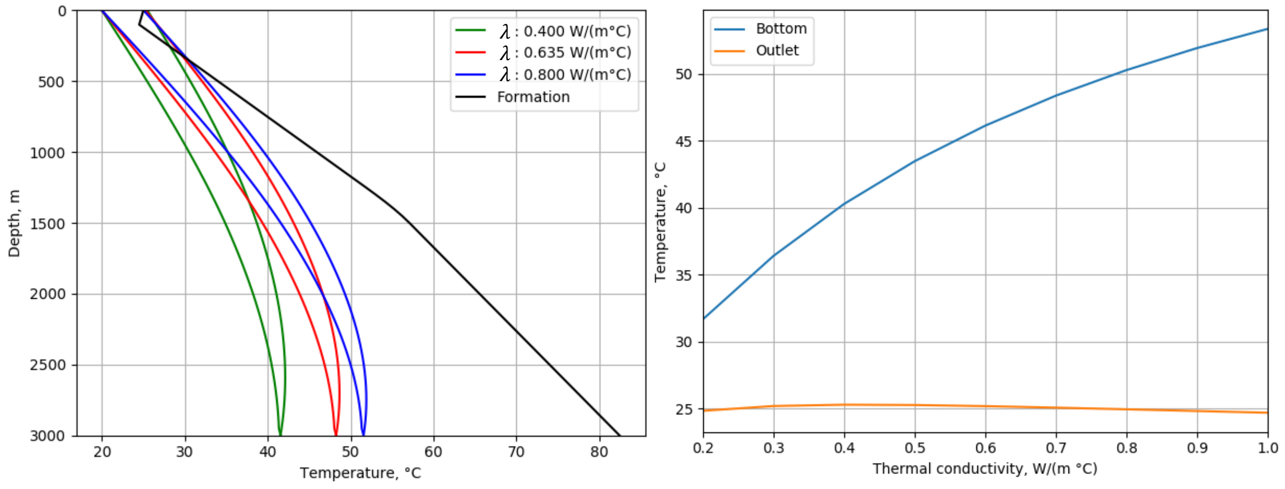


Figure 24. Thermal conductivity effect on temperature profile during drilling

The effect of a thermal conductivity of the drilling fluid with the values 0.4 $\frac{W}{m\cdot°C}$ and 0.8 $\frac{W}{m\cdot°C}$ generates a reduction of $6\,°C$ and an increase of $2\,°C$, respectively. The bottom hole temperature increases together with higher values of thermal conductivity, an increment takes place from 32 $°C$ to 53 $°C$ (66% approximately) due to reduction of the thermal conductivity from 0.2 $\frac{W}{m\cdot°C}$ to 1.0 $\frac{W}{m\cdot°C}$

4.1.5. Density

The drilling fluid density affects directly the flow conditions as shown in the Reynolds number calculation. This parameter is set by defining a reference point at surface conditions, then the entire density profile is calculated based of the pressure and temperature changes along the well. A range from 1000 $\frac{kg}{m^3}$ to 2000 $\frac{kg}{m^3}$ is considered to perform the simulations.

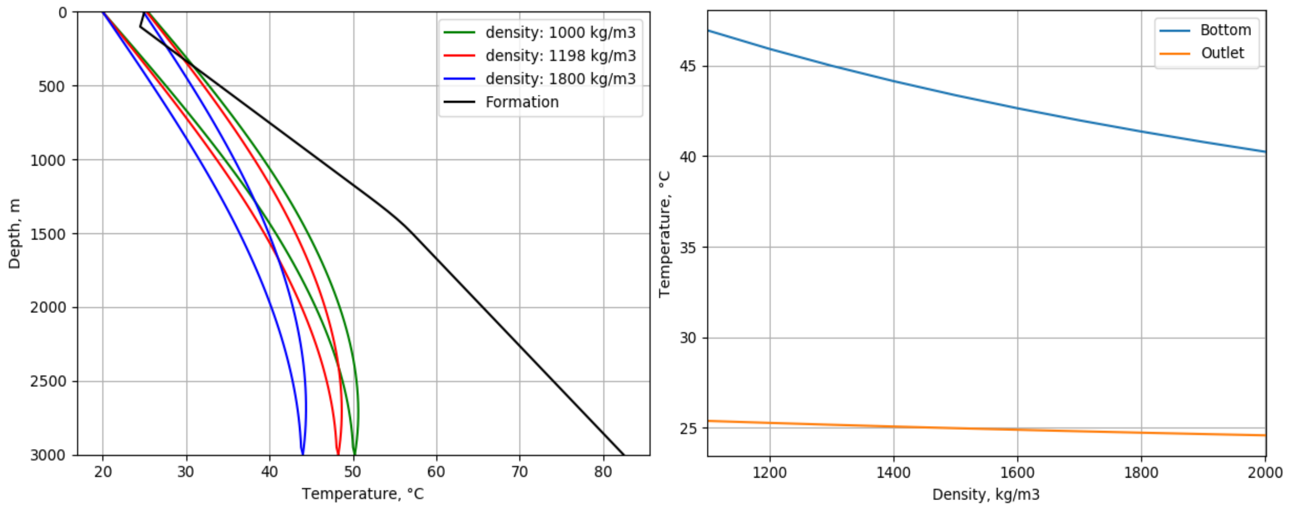


Figure 25. Fluid density effect on temperature profile during drilling

The effect of the mud density on the temperature distribution is very similar to the ones showed due to the flow rate and the specific heat capacity, but more to the last one since the impact is not that significant. The increase from $1198\ \frac{kg}{m^3}$ to $1800\ \frac{kg}{m^3}$ produces a reduction in the bottom hole temperature of only $4\ °C$ approximately (around 8.3%). Furthermore, the temperature at upper sections of the well are barely affected.

4.1.6. Viscosity

The mud viscosity is determined by rheological properties included into the code. However, in order to analyze the effect of this parameter on the temperature profile, a constant value is assumed to perform the simulations in a range of $1\ cP$ to $100\ cP$.

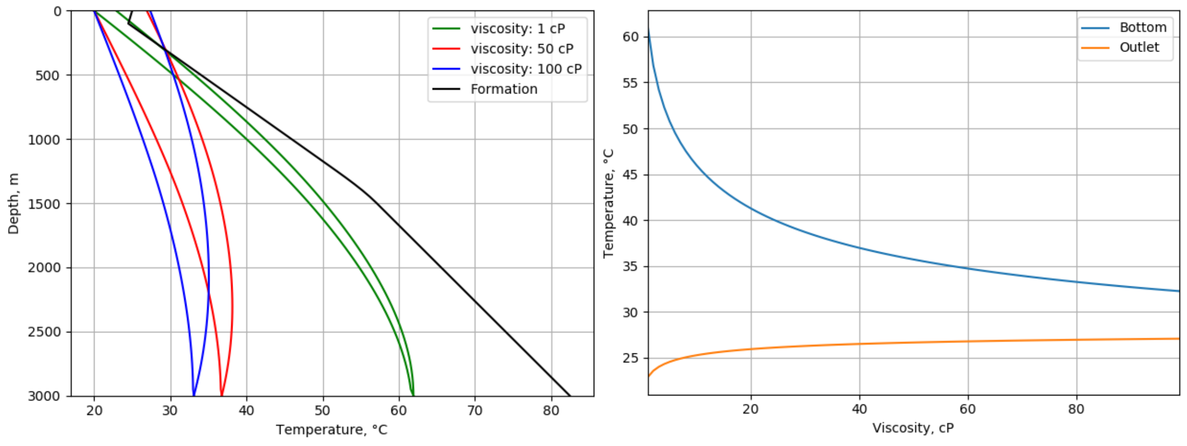


Figure 26. Fluid viscosity effect on temperature profile during drilling

As shown in the figure above, an increment in viscosity from 1 cP to 100 cP decreases the bottom hole temperature around $29\,°C$ ($-47\%\ approximately$). In addition, the temperature difference between the drill pipe and annular fluids is increased as result of the higher viscosity. This parameter acts as a resistance for the heat transfer process, since the viscosity of a fluid is defined as the measure of its resistance to flow and the circulation is an important factor as concluded previously.

4.2. Production

4.2.1. Base case

For the production base case, the surface temperature is set at $25\,°C$. The simulations are performed for 48 $hours$ of operation with a flow rate of 500 $\frac{m^3}{day}$, fluid specific heat capacity of 4000 $\frac{J}{kg{\cdot}°C}$, fluid thermal conductivity of 0.635 $\frac{W}{m{\cdot}°C}$, fluid density of 800 $\frac{kg}{m^3}$ and fluid viscosity of 10 cP.

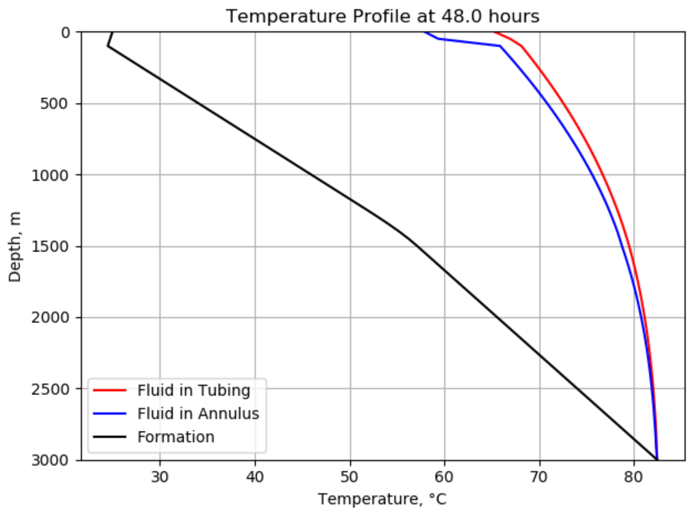


Figure 27. Production base case – temperature profile

4.2.2. Flow rate

In production, the flow rate is also a very important parameter affecting the heat transfer processes. It can magnify the heat exchange through mass transfer, and affects the frictional pressure losses along the tubing. The production rates considered for the simulations are in a range from $500\ \frac{m^3}{day}\ to\ 3000\ \frac{m^3}{day}$.

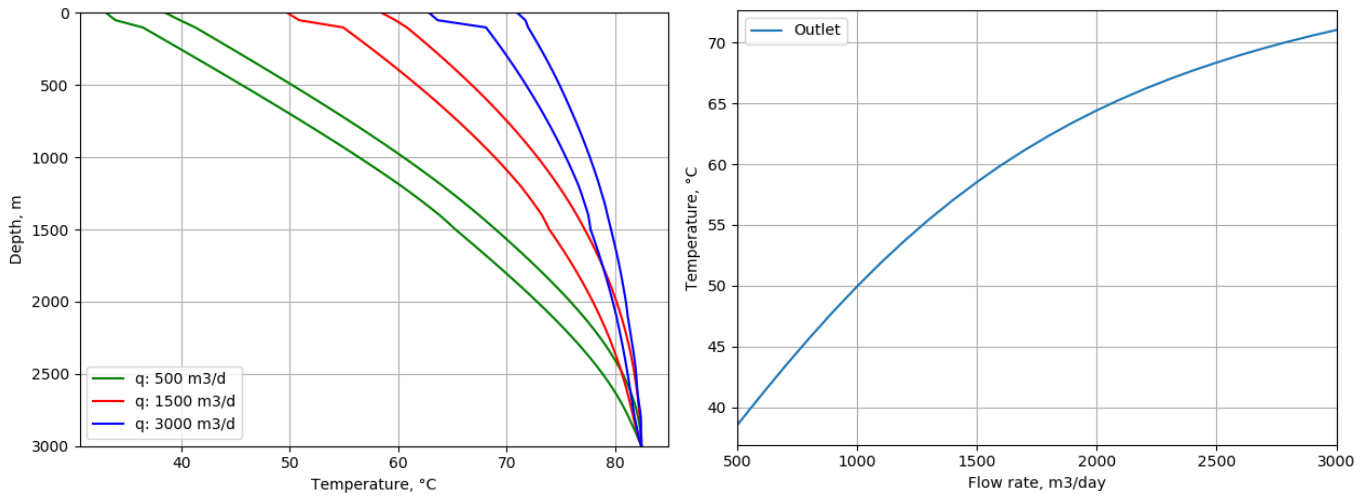


Figure 28. Flow rate effect on temperature profile during production

The results confirm the significant effect of this parameter with the highest impact on the outlet temperature. The change to $1500\ \frac{m^3}{day}$ and $3000\ \frac{m^3}{day}$ produce increases at surface of around $20\ °C$ and $32\ °C$ respectively. The outlet temperature at the tubing top section, increases around 82% due to the variation from the base case up to a flow rate of $3000\ \frac{m^3}{day}$. Therefore, the increase of the circulation rate causes a significant increase on the well temperature profile.

4.2.3. Specific heat capacity

In this case, the specific heat capacity refers to the required amount of heat per unit of mass to increase the produced fluid temperature by one degree. The simulations are performed considering a range from $1000\ \frac{J}{kg{\cdot}°C}$ to $4000\ \frac{J}{kg{\cdot}°C}$.

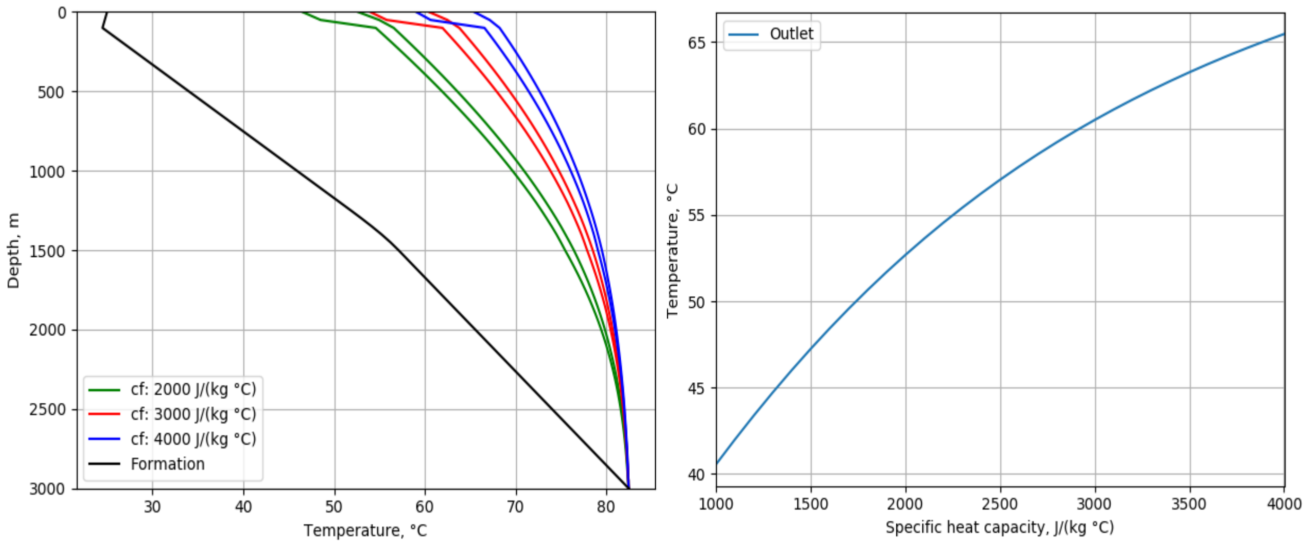


Figure 29. Specific heat capacity effect on temperature profile during production

The results show a considerable increase in the well temperature profile as the specific heat capacity increases. In the production case, the produced fluid is warming the well while circulating upwards through the tubing, a higher value of specific heat capacity means that more heat is required to keep its temperature, thus can transfer the same energy to the surroundings with lower temperature losses. An increment of around $25\ °C$ is produced when the specific heat capacity increases from $1500\ \frac{J}{kg{\cdot}°C}$ to $4000\ \frac{J}{kg{\cdot}°C}$, a change of 60%.

4.2.4. Thermal conductivity

This parameter defines how easily the heat transfer can take place through the production fluid. A range from 0.1 $\frac{W}{m{\cdot}°C}$ to 2.0 $\frac{W}{m{\cdot}°C}$ is implemented to perform the simulations. The thermal conductivity of the completion fluid in the annulus remains constant.

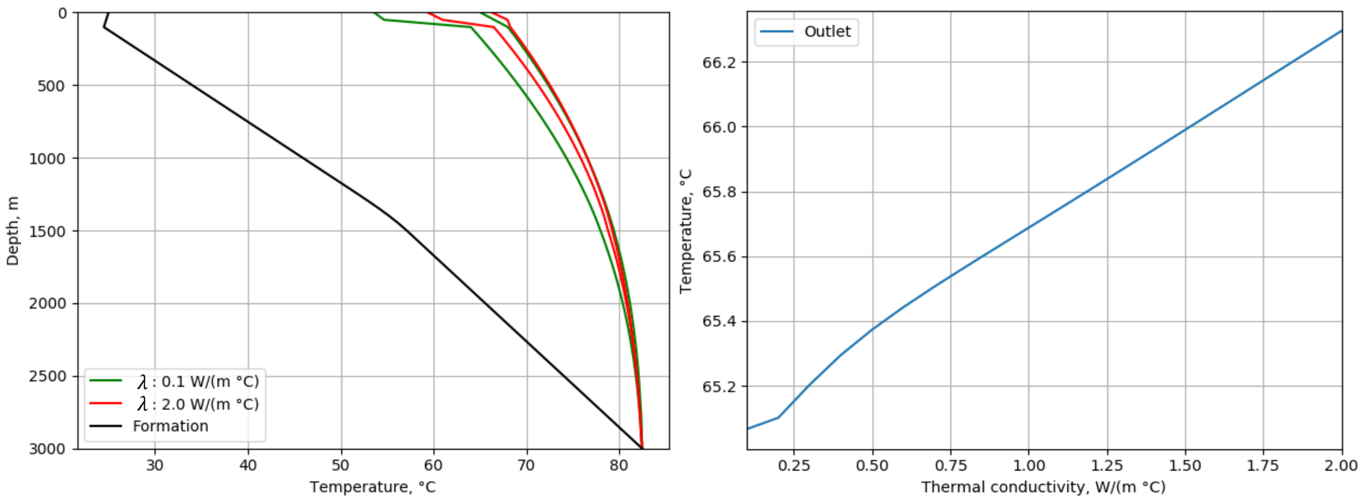


Figure 30. Thermal conductivity effect on temperature profile during production

The results show how the produced fluid temperature barely increases due to the increment on its thermal conductivity. On the other hand, the temperature difference between completion and production fluids is reduced since heat from the last one can easily reach the annular section. The outlet temperature at surface increases only around $1\,°C$ when increasing the thermal conductivity from 0.1 $\frac{W}{m{\cdot}°C}$ to 2.0 $\frac{W}{m{\cdot}°C}$

4.2.5. Density

The produced fluid density is a critical factor for the flow conditions. This parameter is set by defining a reference point at surface conditions, then the entire density profile is calculated based of the pressure and temperature changes along the well. A range from 700 $\frac{kg}{m^3}$ to 900 $\frac{kg}{m^3}$ is considered to perform the simulations.

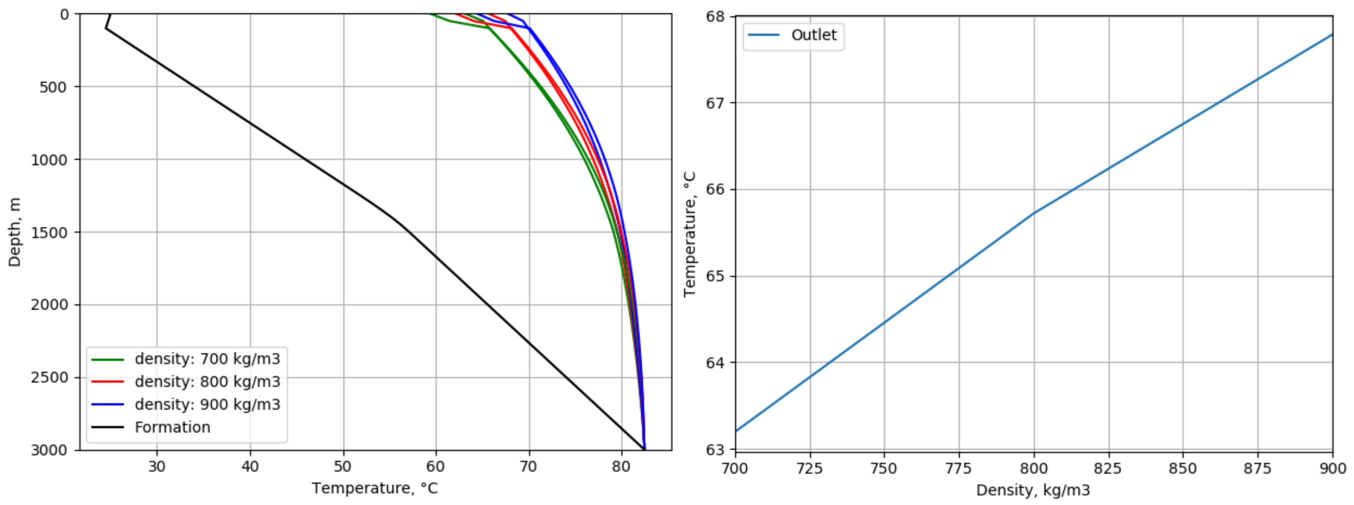


Figure 31. Fluid density effect on temperature profile during production

The outlet temperature at surface presents a higher temperature with a higher density of the production fluid. Lower sections of the well remains constant and the change in temperature starts happening and increasing while going upwards through the tubing and the annular space. The increase from $700\ \frac{kg}{m^3}$ to $900\ \frac{kg}{m^3}$ produces an increment in the outlet temperature of only $4.5\ °C$ approximately (around 7%).

4.2.6. Viscosity

The production fluid viscosity is also closely related to the flow conditions as shown in the Reynolds number calculation. A constant value is assumed to perform the simulations in a range of $10\ cP$ to $5000\ cP$.

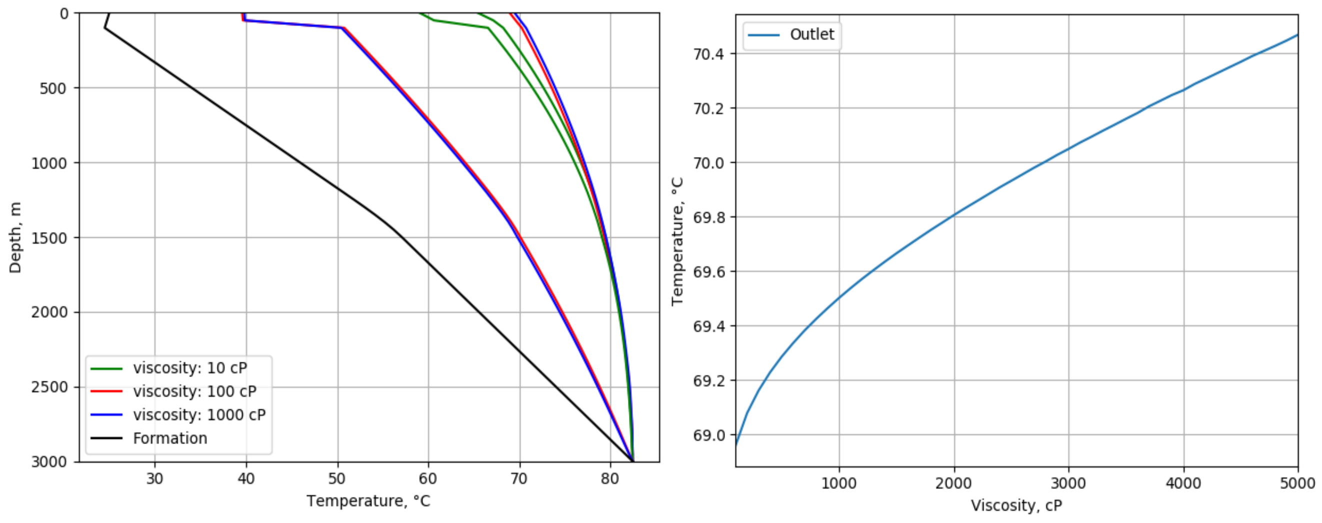


Figure 32. Fluid viscosity effect on temperature profile during production

The production fluid temperature increases a couple of degrees while increasing the viscosity within a range of low values, and then remains almost constant with greater viscosities. On the other hand, the completion fluid temperature at annulus seems to be more affected since higher viscosity values are making more difficult the heat transfer from the produced fluid inside the tubing to the nearby sections. An increment in viscosity from $1000\ cP$ to $5000\ cP$ increases the outlet temperature at surface around only $1\ °C$, but a reduction of $20\ °C$ or more for the annulus temperature at surface

4.3. Injection

4.3.1. Base case

Figure 33 shows the base case temperature profiles of fluid injection scenario. The temperatures in the annulus and in the tubing are nearly the same, which is due to the lower fluid injection rate. For the injection base case, the surface temperature is set at $25\ °C$ and the temperature of the fluid at the drill pipe inlet is $20\ °C$. The simulations are performed for $48\ hours$ of operation with a circulation rate of $300\ \frac{m^3}{day}$, fluid specific heat capacity of $4000\ \frac{J}{kg \cdot °C}$, fluid thermal conductivity of $0.635\ \frac{W}{m \cdot °C}$, fluid density of $1000\ \frac{kg}{m^3}$ and fluid viscosity of $1\ cP$.

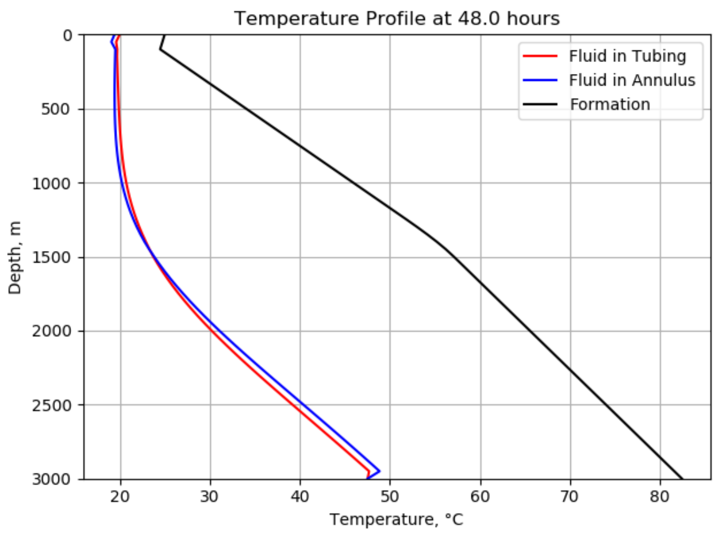


Figure 33. Injection base case – temperature profile

4.3.2. Flow rate

During injection, the flow rate acts similarly as in the production but this time downwards thorugh the tubing. In fact, as during drilling, the injection fluid is cooling down the pipe from the surface. Besides, it magnifies the heat exchange through mass transfer, and affects the frictional pressure losses along the tubing. The simulations are performed with values in the range of $100\ \frac{m^3}{day}\ to\ 500\ \frac{m^3}{day}$. As displayed in Figure 34, results show that as the fluid injection rate decrease the the rate of heat transfer from the formation to the wellbore will increases. During the injection operation, the temperature build up with the respect to the flow rate phenomonon is the opposite of the production operation as shown in Figure 28.

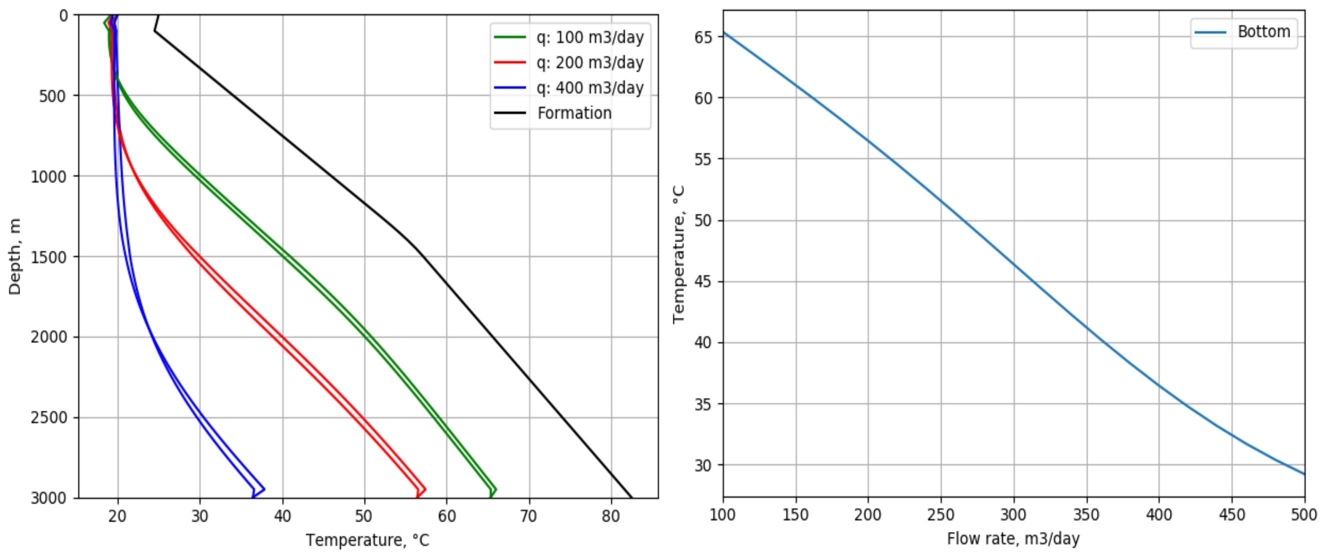


Figure 34. Flow rate effect on temperature profile during injection

The temperature profile seems to be drastically changed because of the injection rate, with the highest impact on the outlet temperature. The increase to $400\ \frac{m^3}{day}$ reduces the bottom hole temperature around $20\ °C$ (-36% approximately), and a flow rate of $100\ \frac{m^3}{day}$ elevates this value around $10\ °C$ ($+18\%$ approximately). The bottom hole temperature decreases around 55% due to the variation from the base case up to a flow rate of $500\ \frac{m^3}{day}$. Therefore, the increase of the circulation rate causes a significant reduction on the well temperature profile.

4.3.3. Specific heat capacity

The effect of changing the specific heat capacity of the injected fluid is analyzed by considering a range from $1000\ \frac{J}{kg{\cdot}°C}$ to $4000\ \frac{J}{kg{\cdot}°C}$. Simulations results are shown in the figure below.

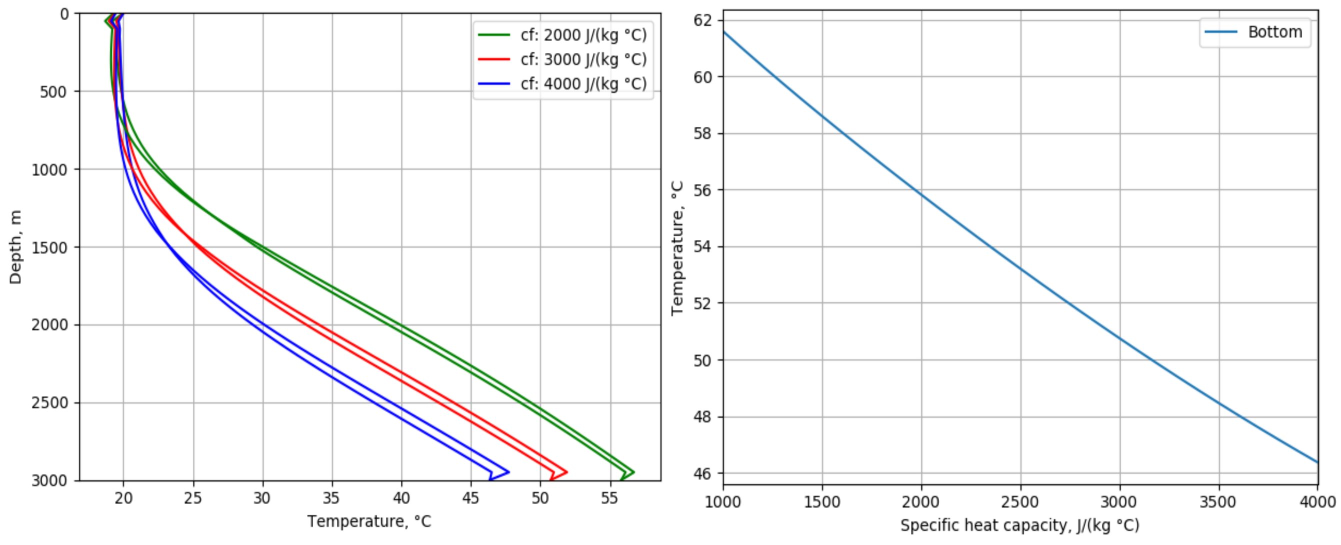


Figure 35. Specific heat capacity effect on temperature profile during injection

During injection, the fluid is receiving heat from the surroundings, similarly as in drilling operations. Therefore, the same amount of energy can increase more the fluid temperature while its specific heat capacity be lower. An increment of around 15 $°C$ is produced when the specific heat capacity is reduced from 4000 $\frac{J}{kg \cdot °C}$ to 1000 $\frac{J}{kg \cdot °C}$, a change of 32%.

4.3.4. Thermal conductivity

This parameter defines how easily the heat transfer can take place through the injected fluid. A range from 0.1 $\frac{W}{m \cdot °C}$ to 2.0 $\frac{W}{m \cdot °C}$ is implemented to perform the simulations. The thermal conductivity of the completion fluid in the annulus remains constant.

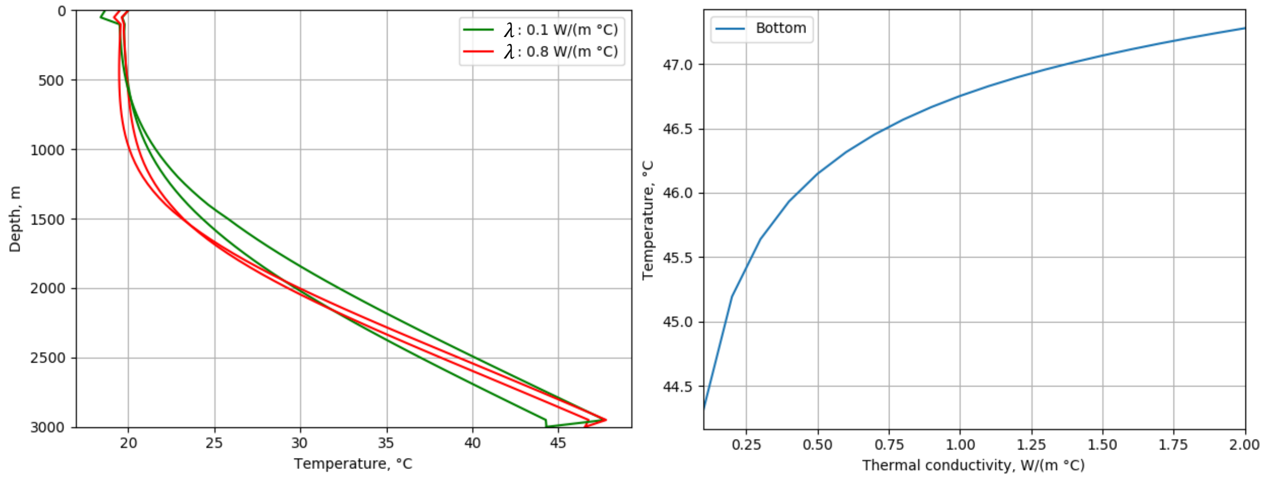


Figure 36. Thermal conductivity effect on temperature profile during injection

The results show how the bottom hole temperature barely increases due to the increment on the thermal conductivity of the injected fluid, while a reduction in the temperature at upper sections takes place. Also, the temperature difference between completion and injection fluids is reduced since heat from the first one can easily reach the tubing section. The bottom hole temperature increases only a value of around $3\,°C$ when increasing the thermal conductivity from 0.1 $\frac{W}{m\cdot°C}$ to 2.0 $\frac{W}{m\cdot°C}$

4.3.5. Density

The injection fluid density affects the circulation behavior and thus the heat process involving the mass transfer from surface to the bottom. This parameter is set by defining a reference point at surface conditions, then the entire density profile is calculated based of the pressure and temperature changes along the well. A range from 1000 $\frac{kg}{m^3}$ to 1500 $\frac{kg}{m^3}$ is considered to perform the simulations.

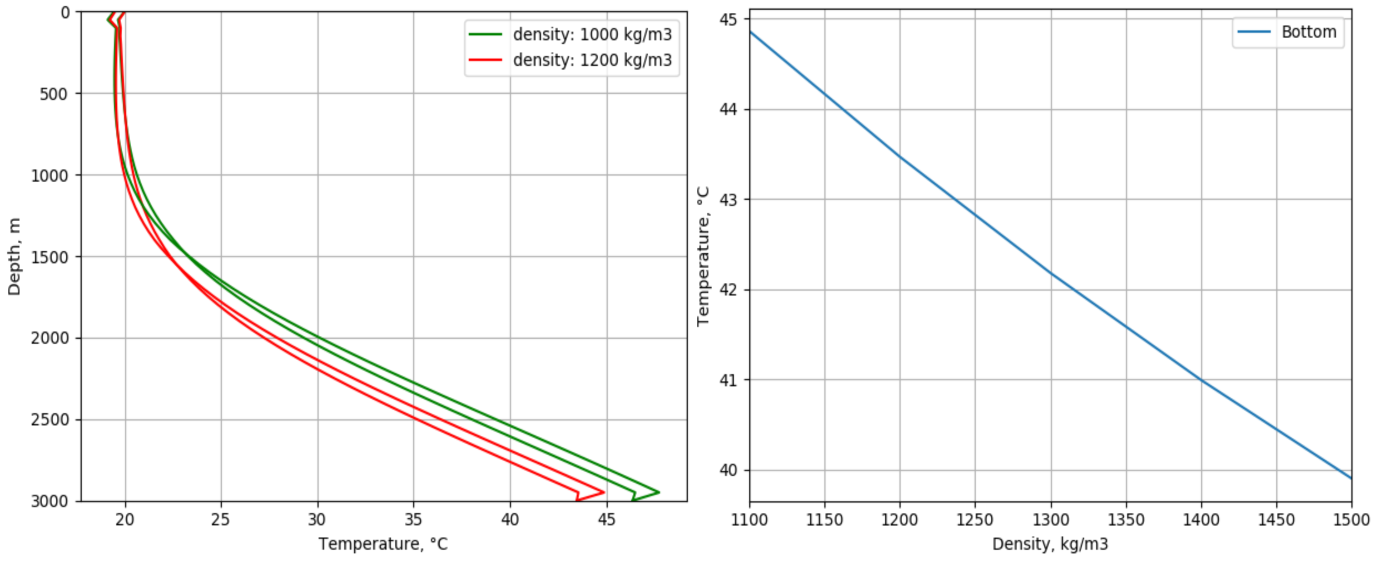


Figure 37. Fluid density effect on temperature profile during injection

The entire temperature profile is reduced due to the effect of injecting a fluid with higher density. The change in temperature grows through depth from the surface up to the bottom of the well. A higher fluid density difficult the reception of thermal energy from the surroundings to the injection fluid inside the tubing. The increase from 1000 $\frac{kg}{m^3}$ to 1500 $\frac{kg}{m^3}$ produces a reduction in the bottom hole temperature of only 6.5 $°C$ approximately (around 14%).

4.3.6. Viscosity

The injection fluid viscosity is also closely related to the flow conditions as shown in the Reynolds number calculation. A constant value is assumed to perform the simulations in a range of 1 cP to 15 cP.

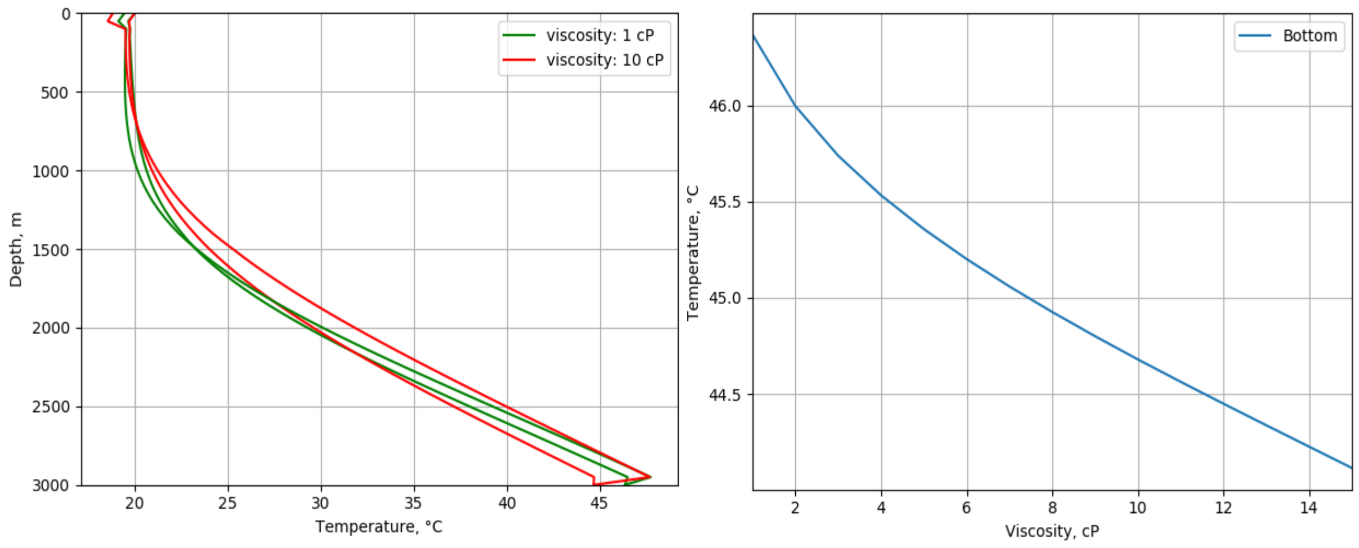


Figure 38. Fluid viscosity effect on temperature profile during injection

The bottom hole temperature decreases a couple of degrees while increasing the fluid viscosity. The middle section presents a more noticeable change compared with the test of the wellbore. Increasing the viscosity also clearly increases the difference between the completion and production fluids temperatures. An increment in viscosity from 1 cP to 15 cP reduced the bottom hole temperature around 2.3 $°C$ (5% approximately).

5. Machine Learning Temperature Predictive Models

Through the years, several equations have been developed in order to capture the physics involved in different operations. Sometimes you need to include a lot of them trying to get higher accuracy. Furthermore, digital approaches have been becoming more popular within the industries, the use of data can reduce the required time to get results and the complexity of certain models when using a vast number of equations.

Simulated data for the three operations included in this thesis (drilling, production and injection) is used to develop respective prediction models. These prediction models are built by using available python packages and the results are compared to evaluate their performances.

Besides, considering that the temperature distributions are defined mainly by the initial condition (formation temperature) despite of the operation type, this work also covers an example of building a prediction model, in this case it is able to predict the formation temperatures for the Norwegian side of the North Sea.

During the development of the models, two methods are considered, “K-nearest neighbors” (KNN) and “LightGBM”. K-NN is defined as: "...an algorithm for classifying n-dimensional objects based on their similarity to the other n-dimensional objects. In machine learning, Nearest Neighbors analysis has been developed as a way to recognize patterns of data without requiring an exact match to any stored patterns or objects. Similar n-dimensional objects are near each other and dissimilar n-dimensional objects are distant from each other. Thus the distance is between two cases is a measure of their dissimilarity."(Parsian, 2015). On the other hand, LightGBM is a gradient boosting decision tree (GBDT) algorithm, which includes Gradient-based One-Side Sampling and Exclusive Feature Bundling as techniques to deal with large number of data instances and large number of features respectively (Ke et al., 2017).

5.1. Prediction of Temperature Profile

Different physics are involved during a certain operation, this leads to a change in the temperature profile along the well, heating or cooling impact will depend on which type of operation is taking place and some parameters and conditions.

5.1.1. Drilling

During a drilling process, cooler mud is circulating downwards through the drill pipe, displacing heated mud by the formation. Therefore, the temperature values decrease while circulation is happening. The figure below shows an example of this effect.

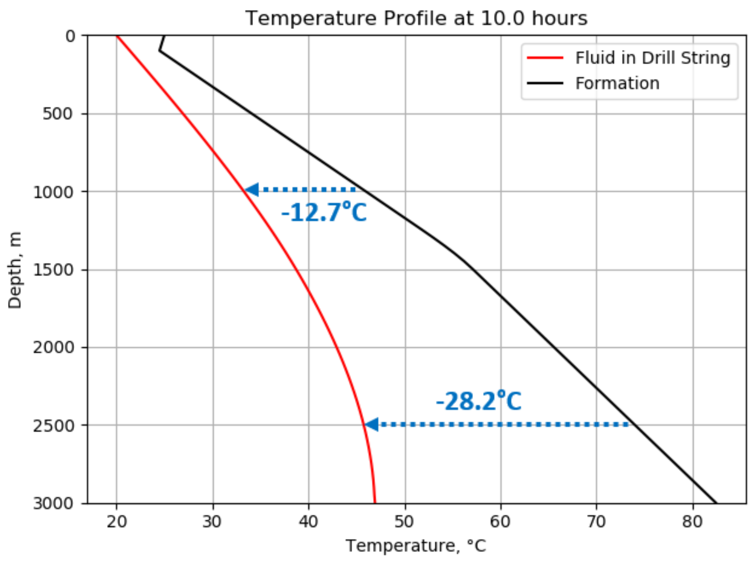
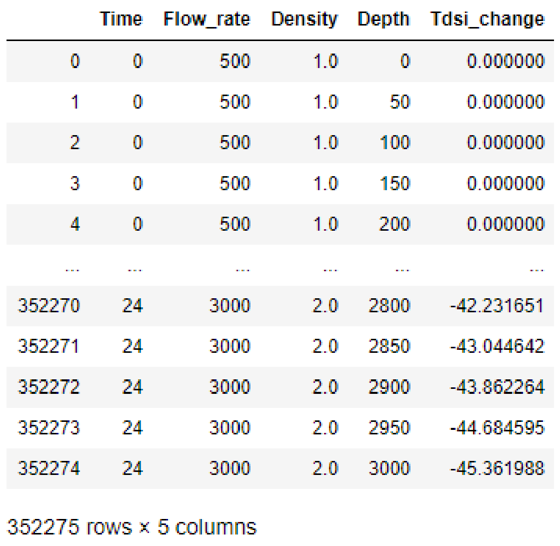


Figure 39. Change in temperature profile during drilling

A total of 5544 cases were simulated considering changes in three main parameters: operation time, circulation rate and mud density. The values considered for this study are as follows:

- Circulation time: up to 24 h, steps of 1 h
- Flow rate: from 500 lpm to 3000 lpm, steps of 250 lpm
- Density: from 1000 kg/m^3 to 2000 kg/m^3, steps of 50 kg/m^3

Data generated for drilling operation is loaded using '*pandas*' package. The model aims to predict the change in temperature from the initial value (formation temperature). Therefore, a new column 'Tdsi_change' to use it as target.



	Time	Flow_rate	Density	Depth	Tdsi_change
0	0	500	1.0	0	0.000000
1	0	500	1.0	50	0.000000
2	0	500	1.0	100	0.000000
3	0	500	1.0	150	0.000000
4	0	500	1.0	200	0.000000
...	...	...	...	...	...
352270	24	3000	2.0	2800	-42.231651
352271	24	3000	2.0	2850	-43.044642
352272	24	3000	2.0	2900	-43.862264
352273	24	3000	2.0	2950	-44.684595
352274	24	3000	2.0	3000	-45.361988

352275 rows × 5 columns

Figure 40. Drilling dataset

Training and testing datasets are obtained by splitting the data in 70% and 30% respectively. Therefore, $246592\ rows$ for training and 105683 for testing.

LightGBM regressor is used to construct a gradient boosting model with the following parameters:

- num_leaves = 5
- learning_rate = 0.05
- n_estimators = 400
- max_bin = 55
- min_data_in_leaf = 2
- min_sum_hessian_in_leaf = 2

The model is then fitted by the training dataset and R^2 is calculated by using r2_score from sklearn.metrics, with a very good value higher than 0.98 as result.

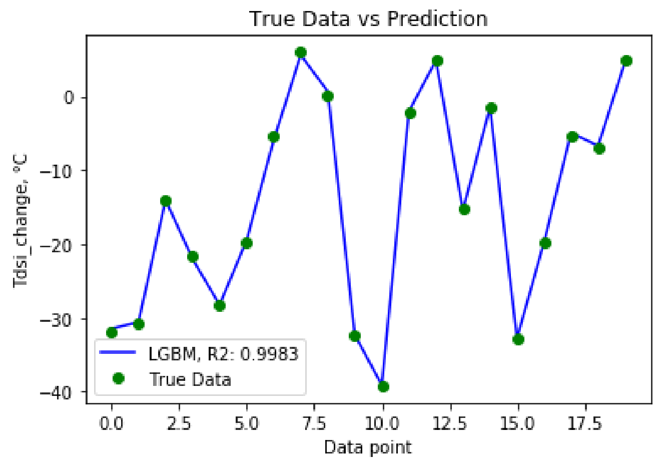


Figure 41. Drilling predictive model evaluation

Once the model shows a good performance with the testing dataset, it is time to evaluate with different parameters not included in the initial dataset, this to analyze if the model can manage new combinations. A drilling process is simulated at $3\ hours$ of operation, the temperature profile is also predicted by using the LGBM model to make a comparison. Main parameters are set as follows: target depth at $3000\ m$, flow rate of $794.933\ lpm$, mud density of $1.198\ sg$, surface temperature of $20°C$ and inlet fluid temperature of $25°C$.

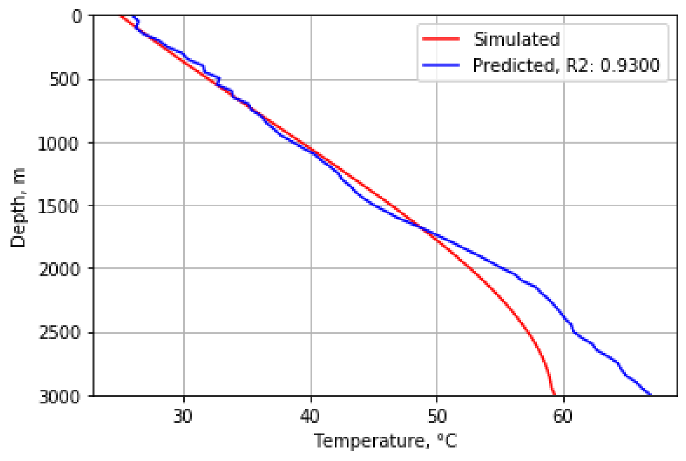


Figure 42. Initial example of temperature predictive model for drilling

The prediction works fine, however, the difference become higher when is close to the bottom. In order to capture more of this behavior, a function can be created to generate a simple correction by applying polynomial regression. This function requires to determine four cases where a variation from each base parameter is considered, the cases are proposed as follows:

Table 4. Parameters cases considered for correction factor calculation (Drilling)

Case	**Target**	**Time, t (h)**	**Flow, q (lpm)**	**Density, ρ (kg/m^3)**
1	Base	1	794.933	1198
2	Time	20	794.933	1198
3	Flow	1	2500	1198
4	Density	1	794.933	1850

Three slopes will compound the correction factor CF based on time, flow and density variations stablished above.

$$m_{time} = (Case_2 - Case_1)/(t_2 - t_1) \tag{89}$$

$$m_{flow} = (Case_3 - Case_1)/(q_2 - q_1) \tag{90}$$

$$m_{density} = (Case_4 - Case_1)/(\rho_2 - \rho_1) \tag{91}$$

$$CF = Case_1 + m_{time}(t - t_1) + m_{flow}(q - q_1) + m_{density}(\rho - \rho_1) \tag{92}$$

The correction factor is integrated then within the prediction in order to get a higher accuracy. The following conditions are considered to test its performance: $t = 16\ h,\ q = 1800\ lpm,$ $\rho = 1300\ kg/m^3$. The figure below shows a comparison between simulated temperature profile and the prediction with the correction factor integrated.

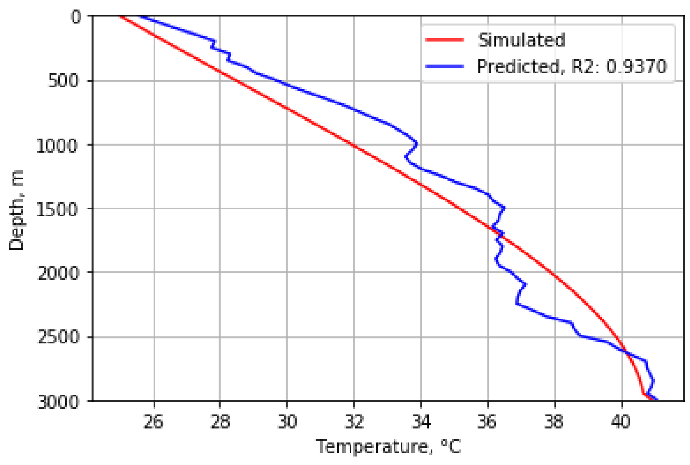


Figure 43. Example of temperature final predictive model for drilling

The predicted profile reproduces the simulation with good accuracy, a R^2 higher than 0.9 confirms that. In addition, the execution time of the simulation is about 1 $second$ per each hour of circulation using 1 min of timestep, while the prediction takes around 3 ms.

5.1.2. Production

During a production process, heated fluid is circulating upwards through the production tubing. Therefore, the temperature difference between the well flowing fluid and the geothermal gradient values increase while fluids are being produced along the well. The figure below shows an example of this effect.

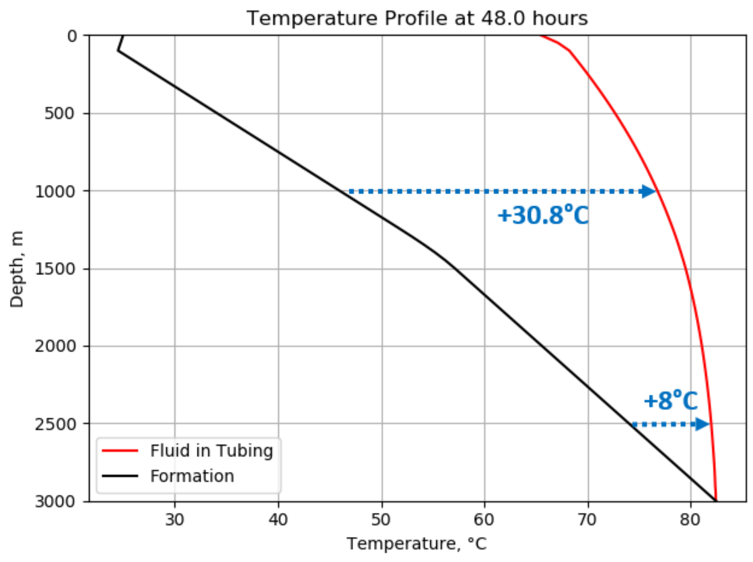
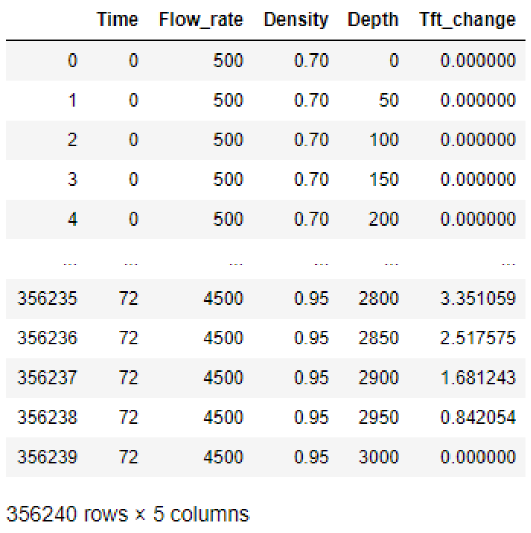


Figure 44. Change in temperature profile during production

A total of 5760 cases were simulated considering changes in three main parameters: operation time, production rate and fluid density. The values considered for this study are as follows:

- Production time: up to 72 h, steps of 1 h
- Production rate: from 500 m^3/day to 3000 m^3/day , steps of 500 m^3/day
- Density: from 700 kg/m^3 to 950 kg/m^3, steps of 50 kg/m^3

Data generated for production operation is loaded using '*pandas'* package. The model aims to predict the change in temperature from the initial value (formation temperature). Therefore, a new column 'Tft_change' to use it as target.



	Time	Flow_rate	Density	Depth	Tft_change
0	0	500	0.70	0	0.000000
1	0	500	0.70	50	0.000000
2	0	500	0.70	100	0.000000
3	0	500	0.70	150	0.000000
4	0	500	0.70	200	0.000000
...	...	...	...	...	...
356235	72	4500	0.95	2800	3.351059
356236	72	4500	0.95	2850	2.517575
356237	72	4500	0.95	2900	1.681243
356238	72	4500	0.95	2950	0.842054
356239	72	4500	0.95	3000	0.000000

356240 rows × 5 columns

Figure 45. Production dataset

Training and testing datasets are obtained by splitting the data in 70% and 30% respectively. Therefore, $249368\ rows$ for training and 106872 for testing.

LightGBM regressor is used to construct a gradient boosting model with the following parameters:

- num_leaves = 5
- learning_rate = 0.05
- n_estimators = 400
- max_bin = 55
- min_data_in_leaf = 2
- min_sum_hessian_in_leaf = 2

The model is then fitted by the training dataset and R^2 is calculated by using r2_score from sklearn.metrics, with a very good value higher than 0.98 as result.

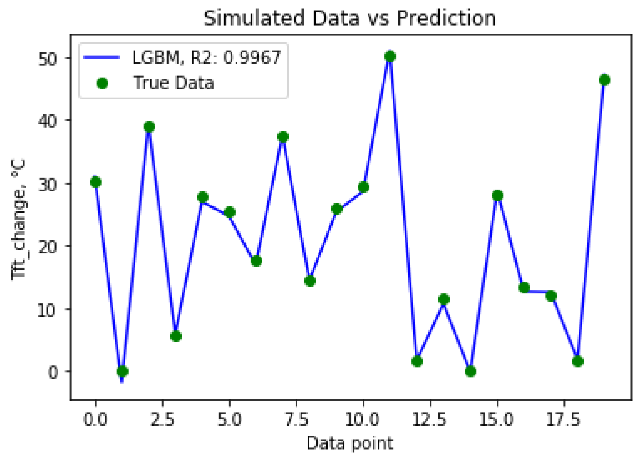


Figure 46. Production predictive model evaluation

Once the model shows a good performance with the testing dataset, it is time to evaluate with different parameters not included in the initial dataset, this to analyze if the model can manage new combinations. A production operation is simulated at 9 $hours$ of operation, the temperature profile is also predicted by using the LGBM model to make a comparison. Main parameters are set as follows: target depth at $3000\ m$, flow rate of $2000\ m^3/day$, fluid density of 0.8 sg and surface temperature of $20°C$.

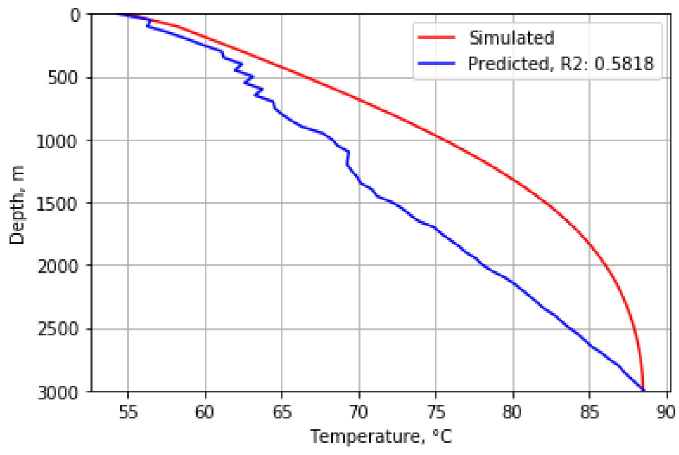


Figure 47. Initial example of temperature predictive model for production

The prediction model tries to copy the trend; however, the difference is significant compared with the simulated data, mainly in the middle of the well. In order to capture more of this behavior, a function can be created to generate a simple correction by applying polynomial regression. This function requires to determine four cases where a variation from each base parameter is considered, the cases are proposed as follows:

Table 5. Parameters cases considered for correction factor calculation (Production)

Case	**Target**	**Time, t (h)**	**Flow, q (m^3/day)**	**Density, ρ (kg/m^3)**
1	Base	1	2000	800
2	Time	60	2000	800
3	Flow	1	4500	800
4	Density	1	2000	950

The correction factor CF is based on time, flow and density variations stablished above. It is integrated then within the prediction in order to get a higher accuracy (Eq.(89)-(92)). The following conditions are considered to test its performance: $t = 12\ h,\ q = 2800\ m^3/day,$ $\rho = 880\ kg/m^3$. The figure below shows a comparison between simulated temperature profile and the prediction with the correction factor integrated.

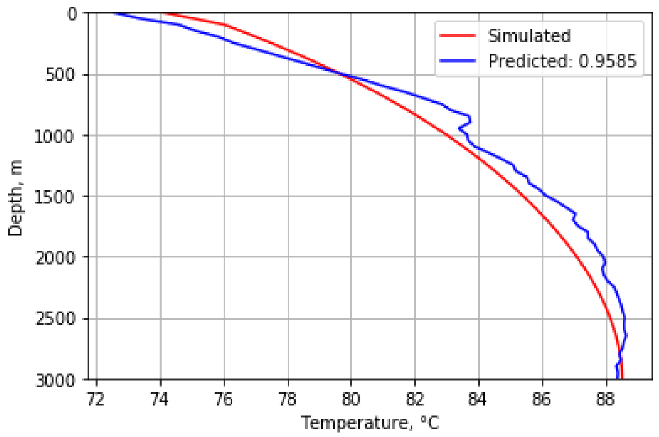


Figure 48. Example of temperature final predictive model for production

The predicted profile reproduces the simulation with good accuracy, a R^2 above 0.95 confirms that. In addition, the execution time of the simulation is about 1 $second$ per each hour of production using 1 min of timestep, while the prediction takes around 4 ms.

5.1.3. Injection

During injection process, similarly as during drilling, cooler mud is circulating downwards through the drill pipe, displacing heated mud by the formation. Unlike production, the temperature difference values decrease while circulation is happening. The figure below shows an example of this effect.

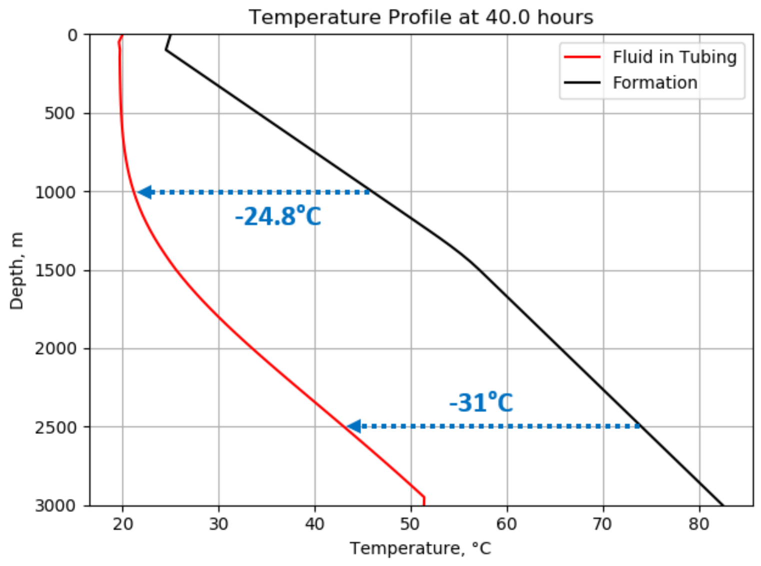
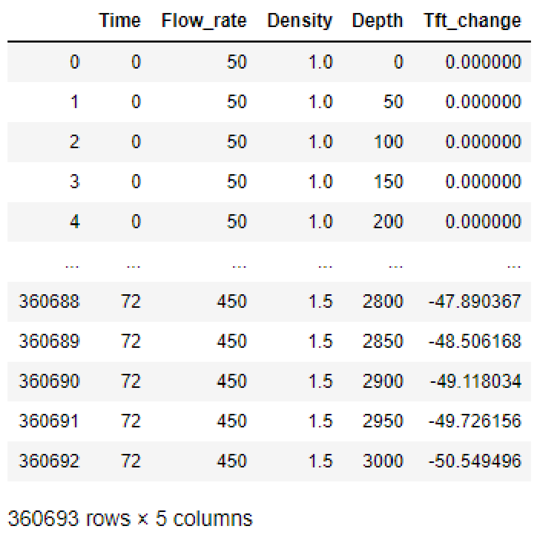


Figure 49. Change in temperature profile during injection

A total of 5832 cases were simulated considering changes in three main parameters: operation time, injection rate and fluid density. The values considered for this study are as follows:

- <u>Injection time:</u> up to 72 h, steps of 1 h
- <u>Injection rate:</u> from 50 m^3/day to 450 m^3/day , steps of 50 m^3/day
- <u>Density:</u> from 1000 kg/m^3 to 1500 kg/m^3, steps of 50 kg/m^3

Data generated for injection operation is loaded using '*pandas'* package. The model aims to predict the change in temperature from the initial value (formation temperature). Therefore, a new column 'Tft_change' to use it as target.



	Time	Flow_rate	Density	Depth	Tft_change
0	0	50	1.0	0	0.000000
1	0	50	1.0	50	0.000000
2	0	50	1.0	100	0.000000
3	0	50	1.0	150	0.000000
4	0	50	1.0	200	0.000000
...	...	...	...	...	...
360688	72	450	1.5	2800	-47.890367
360689	72	450	1.5	2850	-48.506168
360690	72	450	1.5	2900	-49.118034
360691	72	450	1.5	2950	-49.726156
360692	72	450	1.5	3000	-50.549496

360693 rows × 5 columns

Figure 50. Injection dataset

Training and testing datasets are obtained by splitting the data in 70% and 30% respectively. Therefore, 249368 $rows$ for training and 106872 for testing.

LightGBM regressor is used to construct a gradient boosting model with the following parameters:

- num_leaves = 5
- learning_rate = 0.05
- n_estimators = 400
- max_bin = 55
- min_data_in_leaf = 2
- min_sum_hessian_in_leaf = 2

The model is then fitted by the training dataset and R^2 is calculated by using r2_score from sklearn.metrics, with a very good value higher than 0.98 as result.

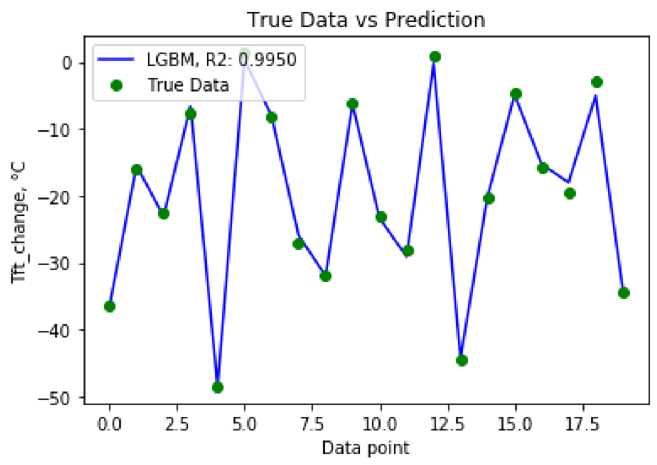


Figure 51. Injection predictive model evaluation

Once the model shows a good performance with the testing dataset, it is time to evaluate with different parameters not included in the initial dataset, this to analyze if the model can manage new combinations. A production operation is simulated at 23 $hours$ of operation, the temperature profile is also predicted by using the LGBM model to make a comparison. Main parameters are set as follows: target depth at 3000 m, flow rate of 144 m^3/day, fluid density of 1.198 sg, surface temperature of 20°C and inlet fluid temperature of 25°C.

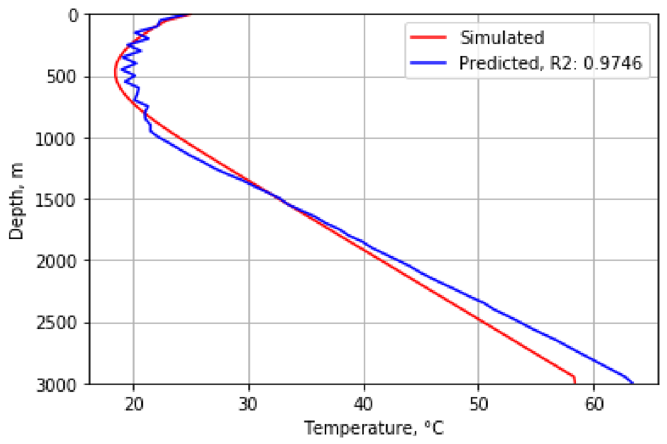


Figure 52. Initial example of temperature predictive model for injection

The prediction works fine, however, the difference become higher when is closer to the bottom. In order to capture more of this behavior, a function can be created to generate a simple correction by applying polynomial regression. This function requires to determine four cases where a variation from each base parameter is considered, the cases are proposed as follows:

Table 6. Parameters cases considered for correction factor calculation (Injection)

Case	Target	Time, t (h)	Flow, q (m^3/day)	Density, ρ (kg/m^3)
1	Base	1	144	1198
2	Time	30	144	1198
3	Flow	1	350	1198
4	Density	1	144	1300

The correction factor CF is based on time, flow and density variations stablished above. It is integrated then within the prediction in order to get a higher accuracy (Eq.(89)-(92)). The following conditions are considered to test its performance: $t = 16\ h,\ q = 260\ m^3/day$, $\rho = 1240\ kg/m^3$. The figure below shows a comparison between simulated temperature profile and the prediction with the correction factor integrated.

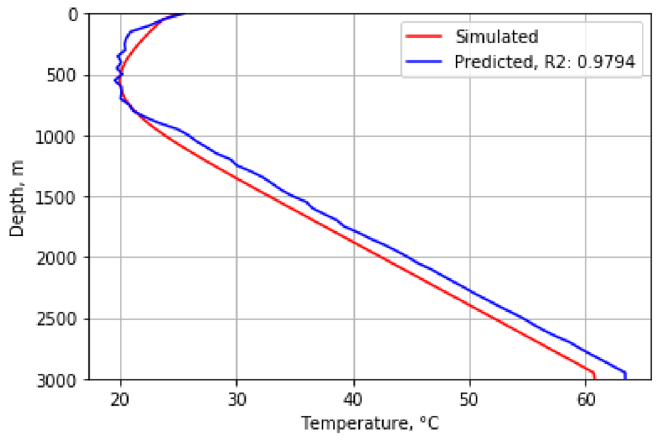


Figure 53. Example of temperature final predictive model for injection

The predicted profile reproduces the simulation with good accuracy, a R^2 higher than 0.97 confirms that. In addition, the execution time of the simulation is about 1 $second$ per each hour of injection using 1 min of timestep, while the prediction takes around 4 ms.

5.2. Prediction of Formation Temperature

There are different operations that a well must deal with, each one of these can generate a heat transfer process along the well with an impact dependent on several factors. Nevertheless, the base of the temperature distribution is always the formation which define the initial conditions and boundaries that will be respected during the total time of the operation.

A good amount of wellbore data is available from the NPD's website. This data is collected from different activities developed by petroleum companies within the Norwegian nation and it is easily accessible through the API. This thesis uses a dataset available from exploration wells, which includes the exact position, water depth, true vertical depth and bottom hole temperature.

The dataset shows information of 1943 wells in North Sea, the Norwegian Sea and the Barents Sea. Figure 54 shows the distribution of these wells on the map.

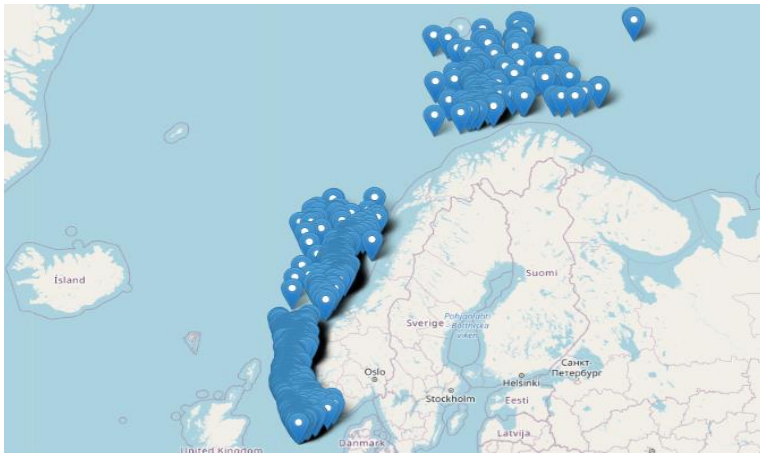


Figure 54. Exploration wells in Norway. Data from NPD

Nevertheless, this work only considers the wells that are located within the North Sea, this means the use of data from 1392 wells in total.

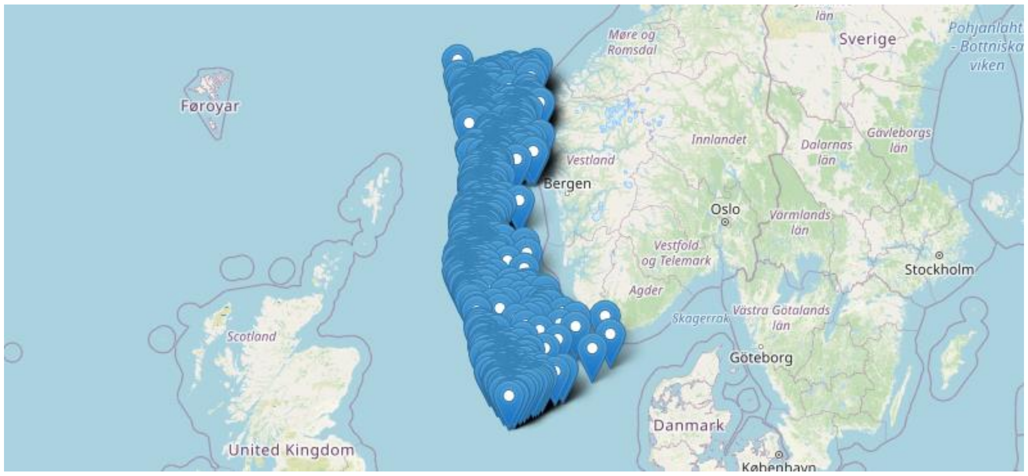


Figure 55. Wells within the North Sea (Norway)

After selecting the raw data, the whole dataset is cleaned and processed. The application of some filters results in 826 wells. Any row with 0 °C of bottom hole temperature is dropped.

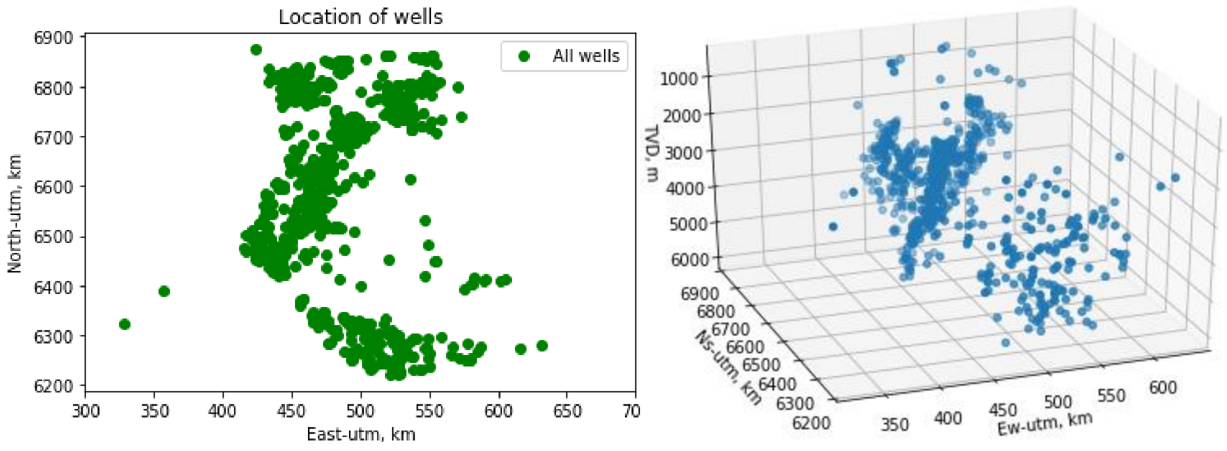


Figure 56. Available wells locations after data cleaning and processing

The dataset was filtered and is ready to start building the prediction model, 70% will be used for the training dataset and 30% for testing, this means 578 and 248 wells, respectively. LGBM package is used again in this section, and additionally, a K-Nearest Neighbors algorithm is performed in order to compare their effectiveness in this case.

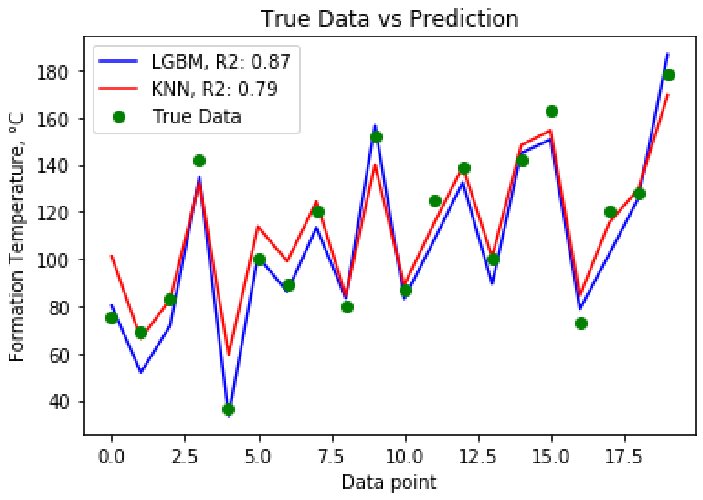


Figure 57. Comparison between LGBM and KNN models performances on formation temperature prediction

The LGBM prediction model allow to get a value of formation temperature for a specific location and a certain depth. Therefore, it is possible to create a square grid within the Norwegian North Sea section and generate the temperature distribution at a defined vertical depth. Figure 58 shows a 3D version of the prediction at 3000 m TVD, this allows to easily check how different can be the conditions in nearby sectors.

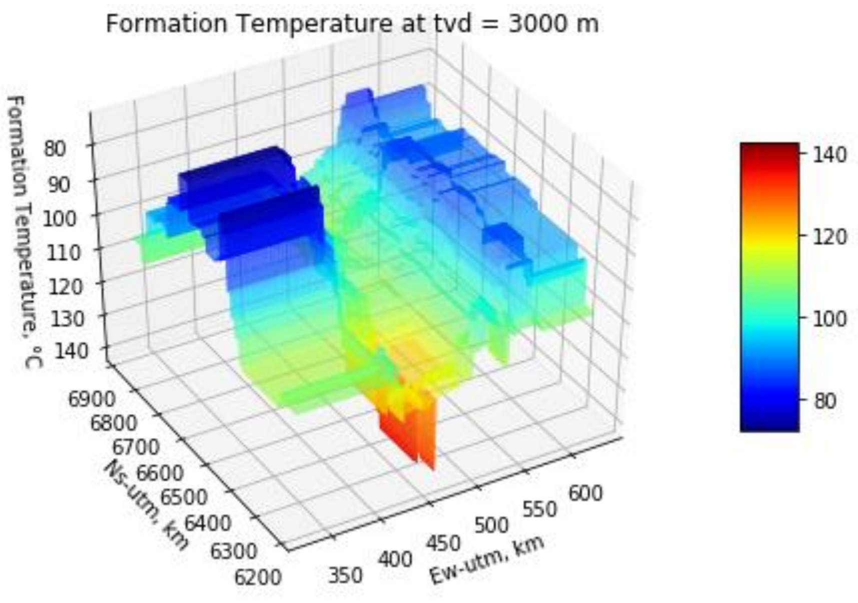


Figure 58. Predicted formation temperature distribution at 3000 m depth – 3D view

Furthermore, other approaches can be implemented to visualize and understand the predicted data. A 2D contour map allow to easily identify those zones where temperature gradients are significantly higher or lower than the average. For instance, the figure below shows a sector among 6600 km – 6700 km, Ns-utm, 350 km – 450 km Ew-utm, where the formation temperature is below 80 °C while the average is around 100 °C, thus, this location presents a low temperature gradient compared with the entire analyzed area.

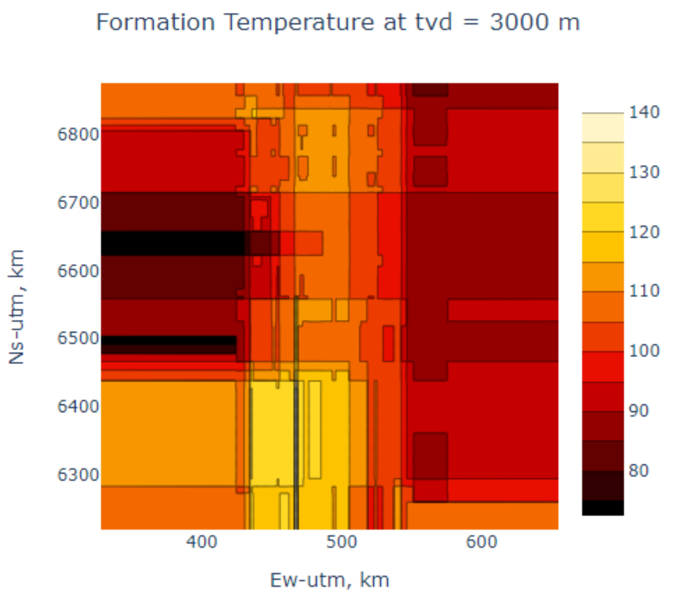


Figure 59. Predicted formation temperature distribution at 3000 m depth – contour map

The prediction model previously developed assumes no difference between location of the well head and location of the bottom hole. In other words, the bottom hole temperature is assigned for the location of the well (well head), assuming no differences in the water depth and temperature gradient. However, this work also consider a second approach, where the difference in position is included.

Raw data with bottom hole locations (see F.2. Loading well bottom locations) was provided allowing to develop and implement another prediction model for the water depth in order to assign a proper estimation of this value to the exact position of the bottom, where the temperature measurement took place. Figure 60 shows the predicted seabed depth within the analyzed area in the North Sea.

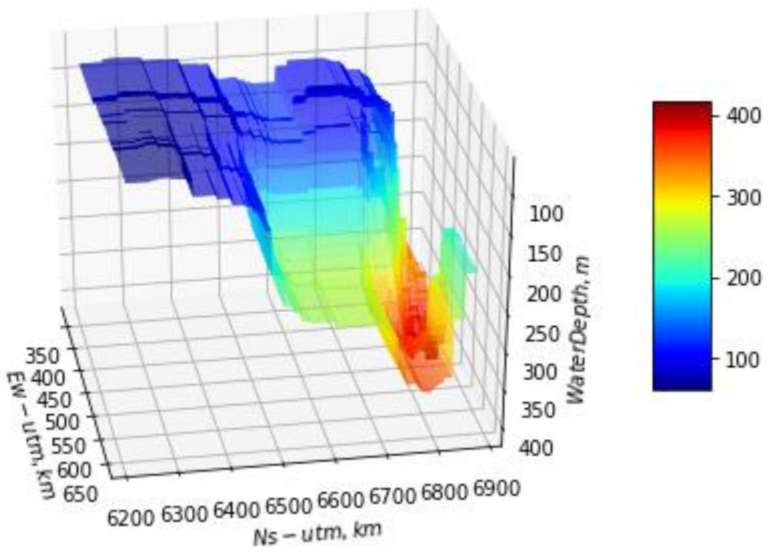


Figure 60. Predicted seabed depth – 3D view

Figure 61 presents a top view of the water depth in the grid.

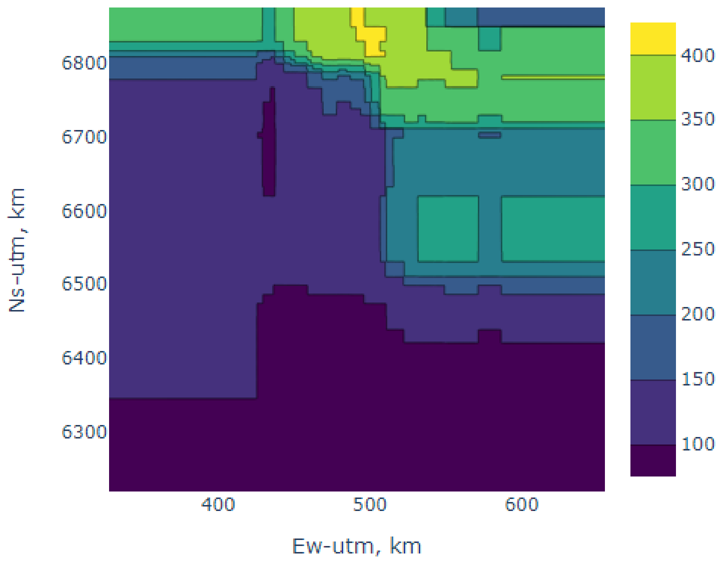


Figure 61. Predicted seabed depth – contour map

The data provided for the development of the water depth prediction model includes the bottom hole location of 472 wells of 1392 considered in the area. Figure 62 shows the wells with this information available (in blue) and the total of wells considered initially (in green).

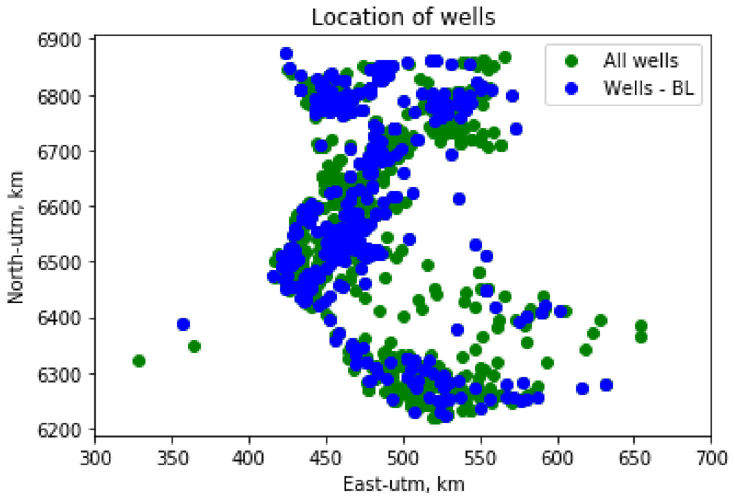


Figure 62. Total wells and wells with available bottom hole location

Moreover, the data is filtered and processed, having a total of 284 wells or rows to keep forward. The wellbores included in the dataset used to fit the prediction model present a variety of target depths from less than $1000\ m$ up to more than $6000\ m$. Figure 63 shows the target points of these wells.

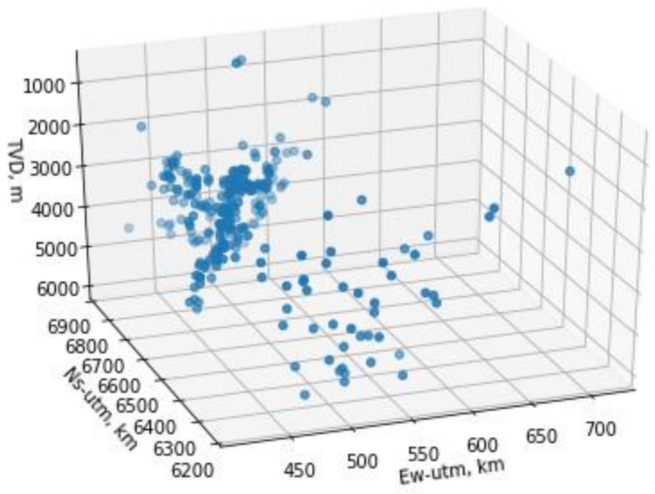


Figure 63. Well target 3D locations

Once the dataset includes target location and vertical depth, it is ready to start the development of the formation temperature prediction model. As the previous case, 70% of the data is used for the training dataset and 30% for testing, this means 198 and 86 wells, respectively. Again, LGBM and K-Nearest Neighbors algorithms are performed and compared.

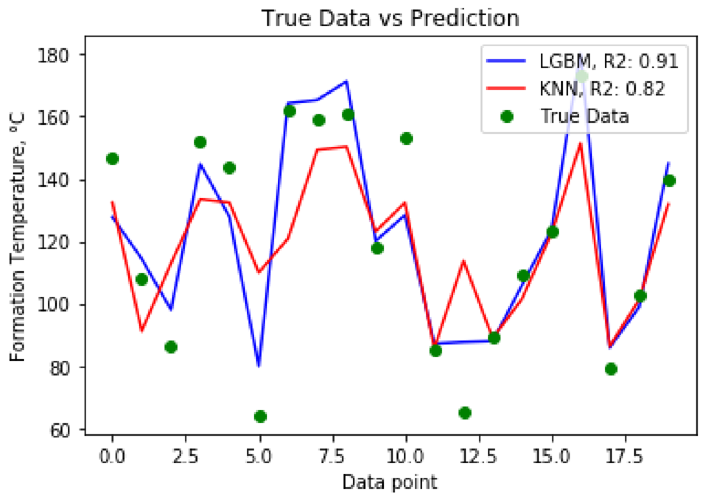


Figure 64. Comparison between LGBM and KNN models performances on formation temperature prediction, second model

This prediction model is used to generate the formation temperature distribution within the study grid at $3000\ m$ TVD.

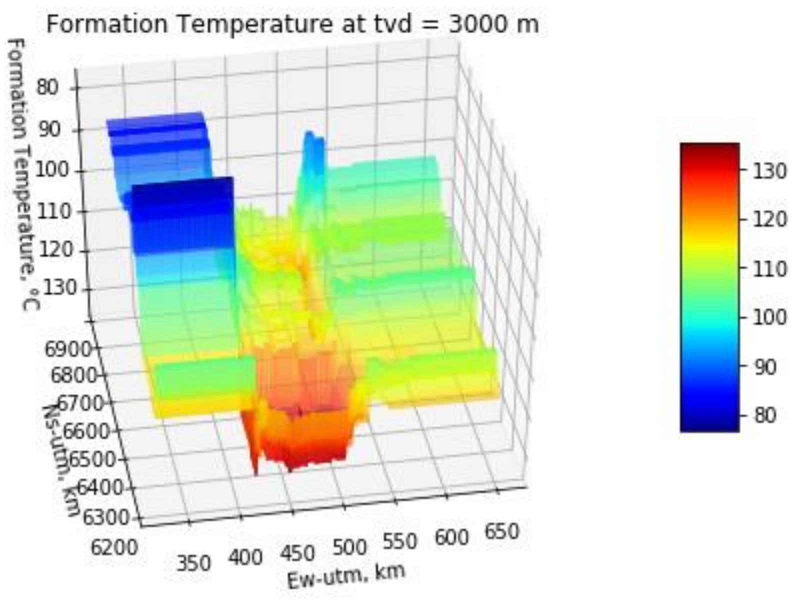


Figure 65. Predicted formation temperature distribution at 3000 m depth – 3D view, second model

6. Conclusions

The produced software tool can simulate successfully the heat transfer through different wellbore configurations during drilling, production or injection. Python, as a high-level programming language, provides several useful tools able to generate a wellbore scenario, discretize it section by section, generate a heat transfer penta-diagonal matrix involving a vast amount of parameters and sub-calculations, solve the entire system and so on, through a very effective manner.

The heat transfer process during a drilling operation is characterized mainly by mud circulation downwards inside the drill pipe and upwards through the annular space, energy propagation from formation to the well through layers of cement tubulars and fluids, and mechanical and hydraulic friction turning into heat along the well. In general, the whole wellbore is cooling down due to the drilling fluid motion despite of heat generation because of the drill pipe rotation.

During drilling, a lot of parameters take part of the heat transfer, which generally influence mainly the bottom hole temperature. Flow rate produces a noticeable impact on the temperature profile, it is significantly reduced while this parameter increases. The same effect occurs while increasing the fluid specific heat capacity, viscosity and density, but less pronounced in the last one. Also, the fluid thermal conductivity reduces the cooling effect while increasing, i.e. the temperature values also increases.

Furthermore, a production process involves fluid circulation through the pipe, but upwards from the formation up to the surface in this case. Similarly, thermal energy is propagated between wellbore and formation; heated produced fluid from the bottom will transfer heat through layers of tubulars and cement to the surrounding space. The entire well is then heating up by production fluids.

During production, the flow rate produces again a significant influence on the temperature distribution along the well, where the highest difference appears at well head. The temperature increases with a higher production rate. In the same way, the increase of the fluid density produces higher thermal values but barely compared with the circulation rate. The increment in fluid viscosity leads to little change in outlet temperature but high increase of the temperature difference between inside the tubing and annular. Besides, thermal conductivity and specific heat capacity show low to practically no effect, inverse and direct respectively.

Moreover, the fluid injection from surface to the reservoir through the tubing, implies that the whole wellbore is cooling down due to the fluid circulation, similarly as during the drilling process. Thermal energy is also propagated in the radial axis through all the layers of tubulars, fluids and cement sheath, from the formation to the wellbore.

During injection, the bottom hole temperature shows to be the most affected point, equally as while drilling. The flow rate and specific heat capacity produce a high inverse impact on the thermal distribution while fluid density and viscosity show low influence in the same direction. In contrast, the fluid thermal conductivity generates little positive effect on the temperature value.

Regarding the prediction models, the algorithms show an accurate and faster (reduction in time of approximately 99.7%) alternative to estimate the temperature profiles when dealing with operations of drilling, production or injection. Nonetheless, only three parameters were considered during the development of these predictive models. Therefore, it is recommended to include more of the parameters in future works in order to get more details about the scope of this methodology.

Likewise, predictive regression algorithms allow to generate successfully a 3D formation temperature map within a selected area if there are measured values inside this. The reliability of the result would highly depend on the density (number of data points / covered volume) and consistency of the measurements used to fit the prediction model. This is very important, considering that the base temperature profile for a wellbore is defined by the formation temperature. For future works, it is recommended to perform this methodology for different areas around the world, where the petroleum activity is strong.

References

Aadnoy, B. S., Fazaelizadeh, M., & Hareland, G. (2010). A 3D Analytical Model for Wellbore Friction. *Journal of Canadian Petroleum Technology*, *49*(10), 25–36. https://doi.org/10.2118/141515-PA

Adewole, J., & Najimu, M. (2017). A Study on the Effects of Date Pit-Based Additive on the Performance of Water-Based Drilling Fluid. *Journal of Energy Resources Technology*, *140*. https://doi.org/10.1115/1.4038382

Ali, U., Abdullah, F. A., & Ismail, A. I. (2017). Crank-Nicolson finite difference method for two-dimensional fractional sub-diffusion equation. *Journal of Interpolation and Approximation in Scientific Computing*, *2017*(2), 18–29. https://doi.org/10.5899/2017/jiasc-00117

Alves, I. N., Alhanati, F. J. S., & Shoham, O. (1992). A Unified Model for Predicting Flowing Temperature Distribution in Wellbores and Pipelines. *SPE Production Engineering*, *7*(04), 363–367. https://doi.org/10.2118/20632-PA

Alvi, M. A. A., Khalifeh, M., & Agonafir, M. B. (2020). Effect of nanoparticles on properties of geopolymers designed for well cementing applications. *Journal of Petroleum Science and Engineering*, *191*, 107128. https://doi.org/10.1016/j.petrol.2020.107128

Amorin, R., & Broni-Bediako, E. (2010). *Application of Minimum Curvature Method to Wellpath Calculations*. 9.

Bergman, T. L., Incropera, F. P., DeWitt, D. P., & Lavine, A. S. (2011). *Fundamentals of Heat and Mass Transfer*. John Wiley & Sons.

Cartalos, U., & Dupuis, D. (1993, January 1). *An Analysis Accounting for the Combined Effect of Drillstring Rotation and Eccentricity on Pressure Losses in Slimhole Drilling*. SPE/IADC Drilling Conference. https://doi.org/10.2118/25769-MS

Chang, X., Zhou, J., Guo, Y., He, S., Wang, L., Chen, Y., Tang, M., & Jian, R. (2018). Heat Transfer Behaviors in Horizontal Wells Considering the Effects of Drill Pipe Rotation, and Hydraulic and Mechanical Frictions during Drilling Procedures. *Energies*, *11*(9), 2414. https://doi.org/10.3390/en11092414

Craig, S. H. (2003, January 1). *A Multi-Well Review Of Coiled Tubing Force Matching*. SPE/ICoTA Coiled Tubing Conference and Exhibition. https://doi.org/10.2118/81715-MS

Dittus, F. W., & Boelter, L. M. K. (1985). Heat transfer in automobile radiators of the tubular type. *International Communications in Heat and Mass Transfer*, *12*(1), 3–22. https://doi.org/10.1016/0735-1933(85)90003-X

Dropkin, D., & Somerscales, E. (1965). Heat Transfer by Natural Convection in Liquids Confined by Two Parallel Plates Which Are Inclined at Various Angles With Respect to the Horizontal. *Journal of Heat Transfer*, *87*(1), 77–82. https://doi.org/10.1115/1.3689057

Edwardson, M. J., Girner, H. M., Parkison, H. R., Williams, C. D., & Matthews, C. S. (1962). Calculation of Formation Temperature Disturbances Caused by Mud Circulation. *Journal of Petroleum Technology*, *14*(04), 416–426. https://doi.org/10.2118/124-PA

Fan, H., Zhou, H., Wang, G., Peng, Q., & Wang, Y. (2014). Utility Hydraulic Calculation Model of Herschel-Bulkley Rheological Model for MPD Hydraulics. *SPE Asia*

Pacific Oil & Gas Conference and Exhibition. SPE Asia Pacific Oil & Gas Conference and Exhibition, Adelaide, Australia. https://doi.org/10.2118/171443-MS

García, A., Santoyo, E., Espinosa, G., Hernández, I., & Gutiérrez, H. (1998). Estimation of Temperatures in Geothermal Wells During Circulation and Shut-in in the Presence of Lost Circulation. *Transport in Porous Media*, *33*(1), 103–127. https://doi.org/10.1023/A:1006545610080

Glaso, O. (1980). Generalized Pressure-Volume-Temperature Correlations. *Journal of Petroleum Technology*, *32*(05), 785–795. https://doi.org/10.2118/8016-PA

Gnielinski, V. (1976). *New Equations for Heat and Mass Transfer in Turbulent Pipe and Channel Flow*. https://doi.org/null

Haciislamoglu, M., & Langlinais, J. (1990). Non-Newtonian Flow in Eccentric Annuli. *Journal of Energy Resources Technology*, *112*(3), 163–169. https://doi.org/10.1115/1.2905753

Haciislamoglu, Mustafa. (1994, January 1). *Practical Pressure Loss Predictions in Realistic Annular Geometries*. SPE Annual Technical Conference and Exhibition. https://doi.org/10.2118/28304-MS

Hasan, A. R., & Kabir, C. S. (1994). Aspects of Wellbore Heat Transfer During Two-Phase Flow (includes associated papers 30226 and 30970). *SPE Production & Facilities*, *9*(03), 211–216. https://doi.org/10.2118/22948-PA

Hasan, A. Rashid, Kabir, C. S., & Wang, X. (2009). A Robust Steady-State Model for Flowing-Fluid Temperature in Complex Wells. *SPE Production & Operations*, *24*(02), 269–276. https://doi.org/10.2118/109765-PA

Holmes, C. S., & Swift, S. C. (1970). Calculation of Circulating Mud Temperatures. *Journal of Petroleum Technology*, *22*(06), 670–674. https://doi.org/10.2118/2318-PA

Huang, Y., Zheng, W., Zhang, D., & Xi, Y. (2020). A modified Herschel–Bulkley model for rheological properties with temperature response characteristics of poly-sulfonated drilling fluid. *Energy Sources, Part A: Recovery, Utilization, and Environmental Effects*, *42*(12), 1464–1475. https://doi.org/10.1080/15567036.2019.1604861

Johancsik, C. A., Friesen, D. B., & Dawson, R. (1984). Torque and Drag in Directional Wells-Prediction and Measurement. *Journal of Petroleum Technology*, *36*(06), 987–992. https://doi.org/10.2118/11380-PA

Kabir, C. S., Hasan, A. R., Kouba, G. E., & Ameen, M. (1996). Determining Circulating Fluid Temperature in Drilling, Workover, and Well Control Operations. *SPE Drilling & Completion*, *11*(02), 74–79. https://doi.org/10.2118/24581-PA

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., & Liu, T.-Y. (2017). LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 30* (pp. 3146–3154). Curran Associates, Inc. http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf

Keller, H. H., Couch, E. J., & Berry, P. M. (1973). Temperature Distribution in Circulating Mud Columns. *Society of Petroleum Engineers Journal*, *13*(01), 23–30. https://doi.org/10.2118/3605-PA

Kothandaraman, C. P. (2006). An Overview of Heat Transfer. In *Fundamentals of Heat and Mass Transfer* (pp. 1–25). New Age International Ltd. http://ebookcentral.proquest.com/lib/uisbib/detail.action?docID=424088

Kumar, A., & Samuel, R. (2013). Analytical Model To Predict the Effect of Pipe Friction on Downhole Fluid Temperatures. *SPE Drilling & Completion*, *28*(03), 270–277. https://doi.org/10.2118/165934-PA

Langaker, I., & Edvardsen, I. (2015, April 22). *Continuous Inclination Enhances TVD Wellbore Positioning at the Troll Fields*. SPE Bergen One Day Seminar. https://doi.org/10.2118/173878-MS

Marshall, D. W., & Bentsen, R. G. (1982). A Computer Model to Determine the Temperature Distributions In a Wellbore. *Journal of Canadian Petroleum Technology*, *21*(01). https://doi.org/10.2118/82-01-05

Nguyen, D., Miska, S., & Yu, M. (2010). *Modeling Thermal Effects on Wellbore Stability*. 23.

Nowak, T. J. (1953). The Estimation of Water Injection Profiles From Temperature Surveys. *Journal of Petroleum Technology*, *5*(08), 203–212. https://doi.org/10.2118/953203-G

Parsian, M. (2015). *Data Algorithms: Recipes for Scaling Up with Hadoop and Spark.* O’Reilly Media, Inc.

Petukhov, B. S. (1970). Heat Transfer and Friction in Turbulent Pipe Flow with Variable Physical Properties. In J. P. Hartnett & T. F. Irvine (Eds.), *Advances in Heat Transfer* (Vol. 6, pp. 503–564). Elsevier. https://doi.org/10.1016/S0065-2717(08)70153-9

Ramey, H. J. J. (1962). Wellbore Heat Transmission. *Journal of Petroleum Technology*, *14*(04), 427–435. https://doi.org/10.2118/96-PA

Raymond, L. R. (1969). Temperature Distribution in a Circulating Drilling Fluid. *Journal of Petroleum Technology*, *21*(03), 333–341. https://doi.org/10.2118/2320-PA

Reid, R. C., Prausnitz, J. M., & Poling, B. E. (1987). *The properties of gases and liquids*. https://www.osti.gov/biblio/6504847

Sagar, R., Doty, D. R., & Schmidt, Z. (1991). Predicting Temperature Profiles in a Flowing Well. *SPE Production Engineering*, *6*(04), 441–448. https://doi.org/10.2118/19702-PA

Samuel, R. (2010). Friction factors: What are they for torque, drag, vibration, bottom hole assembly and transient surge/swab analyses? *Journal of Petroleum Science and Engineering*, *73*(3), 258–266. https://doi.org/10.1016/j.petrol.2010.07.007

Sanni, M. (2018). *Petroleum Engineering: Principles, Calculations, and Workflows*. John Wiley & Sons.

Santoyo, E., Garcia, A., Espinosa, G., Santoyo-Gutiérrez, S., & González-Partida, E. (2003). Convective heat-transfer coefficients of non-Newtonian geothermaldrilling fluids. *Journal of Geochemical Exploration*, *78–79*, 249–255. https://doi.org/10.1016/S0375-6742(03)00146-8

Santoyo-Gutierrez, E. R. (1997). *Transient numerical simulation of heat transfer processes during drilling of geothermal wells* [Phd, University of Salford]. http://usir.salford.ac.uk/id/eprint/14689/

Sieder, E. N., & Tate, G. E. (1936). Heat Transfer and Pressure Drop of Liquids in Tubes. *Industrial & Engineering Chemistry*, *28*(12), 1429–1435. https://doi.org/10.1021/ie50324a027

Stamnes, Ø. N. (2011). *Nonlinear Estimation with Applications to Drilling*. Skipnes. https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/260303

Willhite, G. P. (1967). Over-all Heat Transfer Coefficients in Steam And Hot Water Injection Wells. *Journal of Petroleum Technology*, *19*(05), 607–615. https://doi.org/10.2118/1449-PA

Wooley, G. R. (1980). Computing Downhole Temperatures in Circulation, Injection, and Production Wells. *Journal of Petroleum Technology*, *32*(09), 1.509-1.522. https://doi.org/10.2118/8441-PA

Yang, M., Li, X., Deng, J., Meng, Y., & Li, G. (2015). Prediction of wellbore and formation temperatures during circulation and shut-in stages under kick conditions. *Energy*, *91*, 1018–1029. https://doi.org/10.1016/j.energy.2015.09.001

Zhang, Z., Xiong, Y., & Guo, F. (2018). Analysis of Wellbore Temperature Distribution and Influencing Factors During Drilling Horizontal Wells. *Journal of Energy Resources Technology*, *140*(9), 092901. https://doi.org/10.1115/1.4039744

Zhao, X., Qiu, Z., Wang, M., Huang, W., & Zhang, S. (2018). Performance Evaluation of a Highly Inhibitive Water-Based Drilling Fluid for Ultralow Temperature Wells. *Journal of Energy Resources Technology*, *140*, 012906. https://doi.org/10.1115/1.4037712

Appendix A Wellpath Module

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/wellpath.py

```
def get(mdt, grid_length=50, profile='V', build_angle=1, kop=0, eob=0, sod=0, eod=0, kop2=0, eob2=0, units='metric'):
    """
    Generate a wellpath.
    :param mdt: target depth, m or ft
    :param grid_length: cell's length, m or ft
    :param profile: 'V' for vertical, 'J' for J-type, 'S' for S-type, 'H1' for Horizontal single curve and 'H2' for
                                              Horizontal double curve
    :param build_angle: building angle, °
    :param kop: kick-off point, m or ft
    :param eob: end of build, m or ft
    :param sod: start of drop, m or ft
    :param eod: end of drop, m or ft
    :param kop2: kick-off point 2, m or ft
    :param eob2: end of build 2, m or ft
    :param units: 'metric' or 'english'
    :return: a wellpath object with 3D position
    """

    from numpy import arange
    from math import radians, sin, cos, degrees, acos

    deltaz = 1
    md = list(arange(0, mdt + deltaz, deltaz))  # Measured Depth from RKB, m
    zstep = len(md)  # Number of cells from RKB up to the bottom
    if profile == 'V':        # Vertical well
        tvd = md   # True Vertical Depth from RKB, m
        north = [0] * zstep  # x axis
        east = [0] * zstep  # x axis
        inclination = [0] * zstep
        azimuth = [0] * zstep

    if profile == 'J':        # J-type well
        # Vertical section
        tvd = md[:round(kop / deltaz) + 1]  # True Vertical Depth from RKB, m
        north = [0] * len(tvd)   # x axis
        east = [0] * len(tvd)   # x axis
        inclination = [0] * len(tvd)
        azimuth = [0] * len(tvd)

        # Build section
        s = deltaz
        theta_delta = radians(build_angle / round((eob - kop) / deltaz))
        theta = theta_delta
        r = s / theta
        z_vertical = tvd[-1]
        z_displacement = (r * sin(theta))
        tvd.append(round(tvd[-1] + z_displacement, 2))

        hz_displacement = r * (1 - cos(theta))
        north.append(round(north[-1] + hz_displacement, 2))
        east.append(0)
        inclination.append(degrees(theta))
        azimuth.append(0)

        for x in range(round((eob - kop) / deltaz)-1):
            theta += theta_delta
            inclination.append(degrees(theta))

            z_displacement = (r * sin(theta))
            tvd.append(round(z_vertical + z_displacement, 2))

            hz_displacement = r * (1 - cos(theta)) - north[-1]
            north.append(round(north[-1] + hz_displacement, 2))
            east.append(0)
            azimuth.append(0)

        # Tangent section
        z_displacement = (deltaz * cos(radians(build_angle)))
        hz_displacement = (deltaz * sin(radians(build_angle)))
        for x in range(round((mdt-eob)/deltaz)):
            tvd.append(round(tvd[-1] + z_displacement, 2))
            north.append(round(north[-1] + hz_displacement, 2))
            east.append(0)
            inclination.append(inclination[-1])
            azimuth.append(0)

    if profile == 'S':  # S-type well
        # Vertical section
        tvd = md[:round(kop / deltaz) + 1]  # True Vertical Depth from RKB, m
        north = [0] * len(tvd)  # x axis
        east = [0] * len(tvd)  # x axis
        inclination = [0] * len(tvd)
        azimuth = [0] * len(tvd)

        # Build section
        s = deltaz
        theta_delta = radians(build_angle) / round((eob - kop) / deltaz)
        theta = theta_delta
        r = s / theta

        z_displacement = (r * sin(theta))
        tvd.append(round(tvd[-1] + z_displacement, 2))
        z_count = z_displacement
```

```
hz_displacement = r * (1 - cos(theta))
north.append(round(north[-1] + hz_displacement, 2))
east.append(0)
inclination.append(degrees(theta))
azimuth.append(0)

for x in range(round((eob - kop) / deltaz) - 1):
    theta += theta_delta
    inclination.append(degrees(theta))
    z_displacement = (r * sin(theta)) - z_count
    tvd.append(round(tvd[-1] + z_displacement, 2))
    z_count += z_displacement

    hz_displacement = r * (1 - cos(theta)) - north[-1]
    north.append(round(north[-1] + hz_displacement, 2))
    east.append(0)
    azimuth.append(0)

# Tangent section
z_displacement = (deltaz * cos(radians(build_angle)))
hz_displacement = (deltaz * sin(radians(build_angle)))

for x in range(round((sod - eob) / deltaz)):
    tvd.append(round(tvd[-1] + z_displacement, 2))
    north.append(round(north[-1] + hz_displacement, 2))
    east.append(0)
    inclination.append(inclination[-1])
    azimuth.append(0)

# Drop section
s = deltaz
cells_drop = round((eod - sod) / deltaz)
theta_delta = radians(build_angle) / cells_drop
theta = radians(build_angle)
r = s / theta_delta
z_checkpoint = tvd[-1]
hz_checkpoint = north[-1]
for x in range(cells_drop):
    z_displacement = r * (sin(theta) - sin(theta - (theta_delta * (x + 1))))
    tvd.append(round(z_checkpoint + z_displacement, 2))

    hz_displacement = r * (1 - cos(theta)) - r * (1 - cos(theta - (theta_delta * (x + 1))))
    north.append(round(hz_checkpoint + hz_displacement, 2))
    east.append(0)
    inclination.append(inclination[-1] - degrees(theta_delta))
    azimuth.append(0)

# Vertical section
for x in range(round((mdt - eod) / deltaz)):
        tvd.append(round(tvd[-1] + deltaz, 2))
        north.append(north[-1])  # x axis
        east.append(0)
        inclination.append(0)
        azimuth.append(0)

if profile == 'H1':        # Horizontal single-curve well
    # Vertical section
    tvd = md[:round(kop / deltaz) + 1]  # True Vertical Depth from RKB, m
    north = [0] * len(tvd)  # x axis
    east = [0] * len(tvd)  # x axis
    inclination = [0] * len(tvd)
    azimuth = [0] * len(tvd)

    # Build section
    s = deltaz
    theta_delta = radians(90) / round((eob - kop) / deltaz)
    theta = theta_delta
    r = s / theta

    z_displacement = (r * sin(theta))
    tvd.append(round(tvd[-1] + z_displacement, 2))
    z_count = z_displacement

    hz_displacement = r * (1 - cos(theta))
    north.append(round(north[-1] + hz_displacement, 2))
    east.append(0)
    inclination.append(degrees(theta))
    azimuth.append(0)

    for x in range(round((eob - kop) / deltaz)-1):
        theta += theta_delta
        z_displacement = (r * sin(theta)) - z_count
        tvd.append(round(tvd[-1] + z_displacement, 2))
        z_count += z_displacement

        hz_displacement = r * (1 - cos(theta)) - north[-1]
        inclination.append(degrees(theta))
        north.append(round(north[-1] + hz_displacement, 2))
        east.append(0)
        azimuth.append(0)

    # Horizontal section
    for x in range(round((mdt-eob)/deltaz)):
        tvd.append(tvd[-1])
        north.append(north[-1] + deltaz)
        east.append(0)
        inclination.append(90)
        azimuth.append(0)
```

```
if profile == 'H2':       # Horizontal double-curve well
    # Vertical section
    tvd = md[:round(kop / deltaz) + 2]  # True Vertical Depth from RKB, m
    north = [0] * len(tvd)  # x axis
    east = [0] * len(tvd)  # x axis
    inclination = [0] * len(tvd)
    azimuth = [0] * len(tvd)

    # Build section
    s = deltaz
    theta_delta = radians(build_angle / round((eob - kop) / deltaz))
    theta = theta_delta
    r = s / theta

    z_displacement = (r * sin(theta))
    tvd.append(round(tvd[-1] + z_displacement, 2))
    z_count = z_displacement

    hz_displacement = r * (1 - cos(theta))
    north.append(round(north[-1] + hz_displacement, 2))
    east.append(0)
    inclination.append(degrees(theta))
    azimuth.append(0)

    for x in range(round((eob - kop) / deltaz)-1):
        theta = theta + theta_delta
        z_displacement = (r * sin(theta)) - z_count
        tvd.append(round(tvd[-1] + z_displacement, 2))
        z_count += z_displacement

        hz_displacement = r * (1 - cos(theta)) - north[-1]
        inclination.append(degrees(theta))
        north.append(round(north[-1] + hz_displacement, 2))
        east.append(0)
        azimuth.append(0)

    # Tangent section
    z_displacement = (deltaz * cos(radians(build_angle)))
    hz_displacement = (deltaz * sin(radians(build_angle)))
    for x in range(round((kop2-eob)/deltaz)):
        tvd.append(round(tvd[-1] + z_displacement, 2))
        inclination.append(inclination[-1])
        north.append(round(north[-1] + hz_displacement, 2))
        east.append(0)
        azimuth.append(0)

    # Build section 2
    s = deltaz
    build_angle = 90 - build_angle
    cells_drop = round((eob2 - kop2) / deltaz)
    theta_delta = radians(build_angle) / cells_drop
    theta = radians(build_angle)
    r = s / theta_delta
    z_checkpoint = tvd[-1]
    hz_checkpoint = north[-1]

    for x in range(cells_drop):
        hz_displacement = r * (sin(theta) - sin(theta - (theta_delta * (x + 1))))
        north.append(round(hz_checkpoint + hz_displacement, 2))
        inclination.append(inclination[-1] + degrees(theta_delta))
        east.append(0)
        azimuth.append(0)

        z_displacement = r * (1 - cos(theta)) - r * (1 - cos(theta - (theta_delta * (x + 1))))
        tvd.append(round(z_checkpoint + z_displacement, 2))

    # Horizontal section
    for x in range(round((mdt - eob2) / deltaz)):
        tvd.append(tvd[-1])
        north.append(north[-1] + deltaz)
        inclination.append(inclination[-1])
        east.append(0)
        azimuth.append(0)

# Defining type of section
sections = ['vertical', 'vertical']
for z in range(2, len(tvd)):
    delta_tvd = round(tvd[z] - tvd[z - 1], 9)
    if inclination[z] == 0:  # Vertical Section
        sections.append('vertical')
    else:
        if round(inclination[z], 2) == round(inclination[z - 1], 2):
            if delta_tvd == 0:
                sections.append('horizontal')  # Horizontal Section
            else:
                sections.append('hold')  # Straight Inclined Section
        else:
            if inclination[z] > inclination[z - 1]:  # Built-up Section
                sections.append('build-up')
            if inclination[z] < inclination[z - 1]:  # Drop-off Section
                sections.append('drop-off')

md = md[0::grid_length]
tvd = tvd[0::grid_length]
north = north[0::grid_length]
east = east[0::grid_length]
inclination = inclination[0::grid_length]
azimuth = azimuth[0::grid_length]
sections = sections[0::grid_length]
```

```
    dogleg = [0]
    inc = inclination.copy()
    for x in range(1, len(md)):
        dogleg.append(acos(
            cos(radians(inc[x])) * cos(radians(inc[x - 1]))
            - sin(radians(inc[x])) * sin(radians(inc[x - 1])) * (1 - cos(radians(azimuth[x] - azimuth[x - 1])))
        ))
    dogleg = [degrees(x) for x in dogleg]

    class WellDepths(object):
        def __init__(self):
            self.md = md
            self.tvd = tvd
            self.deltaz = grid_length
            self.zstep = len(md)
            self.north = north
            self.east = east
            self.inclination = [round(i, 2) for i in inclination]
            self.dogleg = dogleg
            self.azimuth = azimuth
            self.sections = sections
            if units == 'english':
                self.md = [i * 3.28 for i in md]
                self.tvd = [i * 3.28 for i in tvd]
                self.deltaz = grid_length * 3.28
                self.north = [i * 3.28 for i in north]
                self.east = [i * 3.28 for i in east]

        def plot(self, azim=45, elev=20):
            plot_wellpath(self, azim, elev, units)

    return WellDepths()


def load(data, grid_length=50, units='metric'):
    """
    Load an existing wellpath.
    :param data: dictionary containing wellpath data (md, tvd, inclination and azimuth)
    :param grid_length: cell's length, m or ft
    :param units: 'metric' or 'english'
    :return: a wellpath object with 3D position
    """

    from numpy import interp, arange
    from math import radians, sin, cos, degrees, acos, tan
    md = [x['md'] for x in data]
    tvd = [x['tvd'] for x in data]
    inc = [x['inclination'] for x in data]
    az = [x['azimuth'] for x in data]
    deltaz = grid_length
```

```
if units == 'english':
    deltaz = grid_length * 3.28

md_new = list(arange(0, max(md) + deltaz, deltaz))
tvd_new = [0]
inc_new = [0]
az_new = [0]
for i in md_new[1:]:
    tvd_new.append(interp(i, md, tvd))
    inc_new.append(interp(i, md, inc))
    az_new.append(interp(i, md, az))
zstep = len(md_new)

dogleg = [0]
for x in range(1, len(md_new)):
    dogleg.append(acos(
        cos(radians(inc_new[x])) * cos(radians(inc_new[x - 1]))
        - sin(radians(inc_new[x])) * sin(radians(inc_new[x - 1])) * (1 - cos(radians(az_new[x] - az_new[x - 1])))
    ))

if 'north' and 'east' in data:
    north = [x['north'] for x in data]
    east = [x['east'] for x in data]
    north_new = [0]
    east_new = [0]
    for i in md_new[1:]:
        north_new.append(interp(i, md, north))
        east_new.append(interp(i, md, east))
else:
    north = [0]
    east = [0]
    for x in range(1, len(md_new)):
        delta_md = md_new[x] - md_new[x - 1]
        if dogleg[x] == 0:
            RF = 1
        else:
            RF = tan(dogleg[x] / 2) / (dogleg[x] / 2)
        north_delta = 0.5 * delta_md * (sin(radians(inc_new[x - 1])) * cos(radians(az_new[x - 1]))
                                        + sin(radians(inc_new[x])) * cos(radians(az_new[x]))) * RF
        north.append(north[-1] + north_delta)
        east_delta = 0.5 * delta_md * (sin(radians(inc_new[x - 1])) * sin(radians(az_new[x - 1]))
                                       + sin(radians(inc_new[x])) * sin(radians(az_new[x]))) * RF
        east.append(east[-1] + east_delta)

dogleg = [degrees(x) for x in dogleg]

# Defining type of section
sections = ['vertical', 'vertical']
for z in range(2, len(tvd_new)):
        delta_tvd = round(tvd_new[z] - tvd_new[z - 1], 9)
        if inc_new[z] == 0:  # Vertical Section
            sections.append('vertical')
        else:
            if round(inc_new[z], 2) == round(inc_new[z - 1], 2):
                if delta_tvd == 0:
                    sections.append('horizontal')  # Horizontal Section
                else:
                    sections.append('hold')  # Straight Inclined Section
            else:
                if inc_new[z] > inc_new[z - 1]:  # Built-up Section
                    sections.append('build-up')
                if inc_new[z] < inc_new[z - 1]:  # Drop-off Section
                    sections.append('drop-off')

    class WellDepths(object):
        def __init__(self):
            self.md = md_new
            self.tvd = tvd_new
            self.inclination = inc_new
            self.azimuth = az_new
            self.dogleg = dogleg
            self.deltaz = deltaz
            self.zstep = zstep
            self.north = north
            self.east = east
            self.sections = sections

        def plot(self, azim=45, elev=20):
            plot_wellpath(self, azim, elev, units)

    return WellDepths()


def plot_wellpath(wellpath, azim=45, elev=20, units='metric'):
    """
    Plot a 3D Wellpath.
    :param wellpath: a wellpath object with 3D position,
    :param azim: set horizontal view.
    :param elev: set vertical view.
    :param units: 'metric' or 'english'
    :return: 3D Plot
    """

    import matplotlib.pyplot as plt
    from mpl_toolkits.mplot3d import Axes3D
    fig = plt.figure()
    ax = Axes3D(fig)
    ax.view_init(azim=azim, elev=elev)
    # Plotting well profile (TVD vs Horizontal Displacement)
```

```
ax.plot(xs=wellpath.east, ys=wellpath.north, zs=wellpath.tvd)
if units == 'metric':
    ax.set_xlabel('East, m')
    ax.set_ylabel('North, m')
    ax.set_zlabel('TVD, m')
else:
    ax.set_xlabel('East, ft')
    ax.set_ylabel('North, ft')
    ax.set_zlabel('TVD, ft')
title = 'Well Profile'
ax.set_title(title)
ax.invert_zaxis()
fig.show()
```

Appendix B Drilling Module

https://github.com/pro-well-plan/pwptemp/tree/master/pwptemp/drilling

B.1. Input.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/drilling/input.py

```
def data(casings=[], d_openhole=0.216, units='metric'):
    """
    Parameters involved within the operation calculations
    :param casings: list of dictionaries with casings characteristics (od, id and depth)
    :param d_openhole: diameter of open hole section, m
    :param units: system of units ('metric' or 'english')
    :return: a dictionary with default values for the required parameters
    """

    from numpy import asarray

    dict_met = {'tin': 20.0, 'ts': 15.0, 'wd': 100.0,  'ddi': 4.0, 'ddo': 4.5, 'dri': 17.716, 'dro': 21.0, 'dfm': 80.0,
        'q': 794.933, 'lambdal': 0.635, 'lambdac': 43.3, 'lambdacem': 0.7, 'lambdad': 40.0, 'lambdafm': 2.249,
        'lambdar': 15.49, 'lambdaw': 0.6, 'cl': 3713.0, 'cc': 469.0, 'ccem': 2000.0, 'cd': 400.0, 'cr': 464.0,
        'cw': 4000.0, 'cfm': 800.0, 'rhof': 1.198, 'rhod': 7.8, 'rhoc': 7.8, 'rhor': 7.8, 'rhofm': 2.245,
        'rhow': 1.029, 'rhocem': 2.7, 'gt': 0.0238, 'wtg': -0.005, 'rpm': 100.0, 'tbit': 9, 'wob': 50, 'rop': 30.4,
        'an': 3100.0, 'bit_n': 1.0, 'dp_e': 0.0, 'thao_o': 1.82, 'beta': 44983 * 10 ** 5, 'alpha': 960 * 10 ** -6,
        'k': 0.3832, 'n': 0.7, 'visc': 0}

    dict_eng = {'tin': 68.0, 'ts': 59.0, 'wd': 328.0, 'ddi': 4.0, 'ddo': 4.5, 'dri': 17.716, 'dro': 21.0, 'dfm': 80.0,
          'q': 300, 'lambdal': 1.098, 'lambdac': 74.909, 'lambdacem': 1.21, 'lambdad': 69.2, 'lambdafm': 3.89,
          'lambdar': 26.8, 'lambdaw': 1.038, 'cl': 0.887, 'cc': 0.112, 'ccem': 0.478, 'cd': 0.096, 'cr': 0.1108,
          'cw': 0.955, 'cfm': 0.19, 'rhof': 9.997, 'rhod': 65.09, 'rhoc': 65.09, 'rhor': 65.09, 'rhofm': 18.73,
          'rhow': 8.587, 'rhocem': 22.5, 'gt': 0.00403, 'wtg': -8.47*10**-4, 'rpm': 100.0, 'tbit': 6637,
          'wob': 11240, 'rop': 99.7, 'an': 3100.0, 'bit_n': 1.0, 'dp_e': 0.0, 'thao_o': 1.82, 'beta': 652423,
          'alpha': 5.33 * 10 ** -4, 'k': 0.3832, 'n': 0.7, 'visc': 0}

    if units == 'metric':
        dict = dict_met
    else:
        dict = dict_eng

    if len(casings) > 0:
        od = sorted([x['od'] * 0.0254 for x in casings])
        id = sorted([x['id'] * 0.0254 for x in casings])
        depth = sorted([x['depth'] for x in casings], reverse=True)
        dict['casings'] = [[od[x], id[x], depth[x]] for x in range(len(casings))]
        dict['casings'] = asarray(dict['casings'])
    else:
        dict['casings'] = [[(d_openhole + dict['dro'] * 0.0254), d_openhole, 0]]
        dict['casings'] = asarray(dict['casings'])

    return dict


def info(about='all'):
    """
    Retrieves information about the parameters (description and units)
    :param about: type of parameters
    :return: description and units of parameters
    """

    print("Use the ID of a parameter to change the default value (e.g. tdict['tin']=30 to change the fluid inlet "
          "temperature from the default value to 30° Celsius)")
    print('Notice that the information is provided as follows:' + '\n' +
          'parameter ID: general description, units' + '\n')

    tubular_parameters = 'VALUES RELATED TO TUBULAR SIZES' + '\n' + \
                         'ddi: drill string inner diameter, in' + '\n' + \
                         'ddo: drill string outer diameter, in' + '\n' + \
                         'dri: riser inner diameter, in' + '\n' + \
                         'dro: riser outer diameter, in' + '\n'

    conditions_parameters = 'PARAMETERS RELATED TO SIMULATION CONDITIONS' + '\n' + \
                            'ts: surface temperature, °C or °F' + '\n' + \
                            'wd: water depth, m or ft' + '\n' + \
                            'dfm: undisturbed formation diameter, m or ft' + '\n'

    heatcoeff_parameters = 'PARAMETERS RELATED TO HEAT COEFFICIENTS' + '\n' + \
                           'lambdal: fluid - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdac: casing - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdacem: cement - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdad: drill pipe - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdafm: formation - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdar: riser - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdaw: water - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'cl: fluid - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'cc: casing - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'ccem: cement - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'cd: drill pipe - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'cr: riser - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'cw: water - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'cfm: formation - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'gt: geothermal gradient, °C/m or °F/ft' + '\n' + \
                           'wtg: seawater thermal gradient, °C/m or °F/ft' + '\n'

    densities_parameters = 'PARAMETERS RELATED TO DENSITIES' + '\n' + \
                           'rhof: fluid density, sg or ppg' + '\n' + \
                           'rhod: drill pipe density, sg or ppg' + '\n' + \
                           'rhoc: casing density, sg or ppg' + '\n' + \
                           'rhor: riser density, sg or ppg' + '\n' + \
                           'rhofm: formation density, sg or ppg' + '\n' + \
                           'rhow: seawater density, sg or ppg' + '\n' + \
                           'rhocem: cement density, sg or ppg' + '\n' + \
                           'beta: isothermal bulk modulus, Pa or psi' + '\n' + \
                           'alpha: expansion coefficient, 1/°C or 1/°F' + '\n'

    viscosity_parameters = 'PARAMETERS RELATED TO MUD VISCOSITY' + '\n' + \
                           'thao_o: yield stress, Pa or psi' + '\n' + \
                           'n: flow behavior index, dimensionless' + '\n' + \
                           'k: consistency index, Pa*s^n or psi*s^n' + '\n' + \
                           'visc: fluid viscosity, cp' + '\n'

    operational_parameters = 'PARAMETERS RELATED TO THE OPERATION' + '\n' + \
                     'tin: fluid inlet temperature, °C or °F' + '\n' + \
                     'q: flow rate, lpm or gpm' + '\n' + \
                     'rpm: revolutions per minute' + '\n' + \
                     'tbit: torque on the bit, kN*m or lbf*ft' + '\n' + \
                     'wob: weight on bit, kN or lbf' + '\n' + \
                     'rop: rate of penetration, m/h or ft/h' + '\n' + \
                     'an: area of the nozzles, in^2' + '\n' + \
                     'bit_n: drill bit efficiency' + '\n' + \
                     'dp_e: drill pipe eccentricity' + '\n'

    if about == 'casings':
        print(tubular_parameters)

    if about == 'conditions':
        print(conditions_parameters)

    if about == 'heatcoeff':
        print(heatcoeff_parameters)

    if about == 'densities':
        print(densities_parameters)

    if about == 'operational':
        print(operational_parameters)

    if about == 'viscosity':
        print(viscosity_parameters)

    if about == 'all':
        print(tubular_parameters + '\n' + conditions_parameters + '\n' + heatcoeff_parameters + '\n' +
              densities_parameters + '\n' + viscosity_parameters + '\n' + operational_parameters)


def set_well(temp_dict, depths, visc_eq=True, units='metric'):
    """
```

```
Define properly the parameters and respective values within an object well.
:param temp_dict: dictionary with inputs and default values.
:param depths: wellpath object
:param visc_eq: boolean to use the same viscosity in the pipe and annular
:param units: system of units ('metric' or 'english')
:return: a well object with conditions and parameters defined
"""

from math import pi, log

def wellpath():
    """
    :return: wellpath object
    """
    return depths

class NewWell(object):
    def __init__(self):
        # DEPTH
        self.md = depths.md
        self.tvd = depths.tvd
        self.deltaz = depths.deltaz
        self.zstep = depths.zstep
        self.sections = depths.sections
        self.north = depths.north
        self.east = depths.east
        self.inclination = depths.inclination
        self.dogleg = depths.dogleg
        self.azimuth = depths.azimuth
        if units != 'metric':
            self.md = [i / 3.28 for i in self.md]
            self.tvd = [i / 3.28 for i in self.tvd]
            self.deltaz = self.deltaz / 3.28
            self.north = [i / 3.28 for i in self.north]
            self.east = [i / 3.28 for i in self.east]

        # TUBULAR
        if units == 'metric':
            d_conv = 0.0254   # from in to m
        else:
            d_conv = 0.0254   # from in to m
        self.casings = temp_dict["casings"]  # casings array
        self.ddi = temp_dict["ddi"] * d_conv  # Drill String Inner  Diameter, m
        self.ddo = temp_dict["ddo"] * d_conv   # Drill String Outer Diameter, m
        self.dri = temp_dict["dri"] * d_conv  # Riser diameter Inner Diameter, m
        self.dro = temp_dict["dro"] * d_conv   # Riser diameter Outer Diameter, m

        # CONDITIONS
        if units == 'metric':
            depth_conv = 1     # from m to m
    self.ts = temp_dict["ts"]  # Surface Temperature (RKB), °C
else:
    depth_conv = 1/3.28   # from ft to m
    self.ts = (temp_dict["ts"] - 32) * (5/9)  # Surface Temperature (RKB), from °F to °C
self.wd = temp_dict["wd"] * depth_conv  # Water Depth, m
self.riser = round(self.wd / self.deltaz)  # number of grid cells for the riser
self.dsr = self.casings[0, 0]  # Surrounding Space Inner Diameter, m
self.dsro = sorted([self.dro + 0.03, self.casings[-1, 0] + 0.03])[-1]  # Surrounding Space Outer Diameter, m
self.dfm = temp_dict["dfm"] * d_conv   # Undisturbed Formation Diameter, m

# RADIUS (CALCULATED)
self.r1 = self.ddi / 2  # Drill String Inner  Radius, m
self.r2 = self.ddo / 2  # Drill String Outer Radius, m
self.r3 = self.casings[0, 1] / 2  # Casing Inner Radius, m
self.r3r = self.dri / 2  # Riser Inner Radius, m
self.r4r = self.dro / 2  # Riser Outer Radius, m
self.r4 = self.casings[0, 0] / 2  # Surrounding Space Inner Radius m
self.r5 = self.dsro / 2  # Surrounding Space Outer Radius, m
self.rfm = self.dfm / 2  # Undisturbed Formation Radius, m

# DENSITIES kg/m3
if units == 'metric':
    dens_conv = 1000     # from sg to kg/m3
else:
    dens_conv = 119.83   # from ppg to kg/m3
self.rhof = temp_dict["rhof"] * dens_conv  # Fluid
self.rhod = temp_dict["rhod"] * dens_conv  # Drill Pipe
self.rhoc = temp_dict["rhoc"] * dens_conv  # Casing
self.rhor = temp_dict["rhor"] * dens_conv  # Riser
self.rhocem = temp_dict["rhocem"] * dens_conv  # Cement Sheath
self.rhofm = temp_dict["rhofm"] * dens_conv  # Formation
self.rhow = temp_dict["rhow"] * dens_conv  # Seawater

# OPERATIONAL
if units == 'metric':
    self.tin = temp_dict["tin"]  # Inlet Fluid temperature, °C
    q_conv = 0.06     # from lpm to m^3/h
    an_conv = 1 / 1500  # from in^2 to m^2
    wob_conv = 1   # from kN to kN
    tbit_conv = 1  # from kN*m to kN*m
    rop_conv = 1    # from m/h to m/h
else:
    self.tin = (temp_dict["tin"] - 32) * (5/9)  # Inlet Fluid temperature, from °F to °C
    q_conv = 0.2271   # from gpm to m^3/h
    an_conv = 1 / 1500  # from in^2 to m^2
    wob_conv = 4.4482 / 1000  # from lbf to kN
    tbit_conv = 1.356 / 1000  # from lbf*ft to kN*m
    rop_conv = 1/3.28  # from ft/h to m/h

self.q = temp_dict["q"] * q_conv  # Flow rate, m^3/h
self.va = (self.q / (pi * ((self.r3 ** 2) - (self.r2 ** 2)))) / 3600  # Fluid velocity through the annular
self.vp = (self.q / (pi * (self.r1 ** 2))) / 3600  # Fluid velocity through the drill pipe
self.rpm = temp_dict["rpm"]  # Revolutions per minute
self.tbit = temp_dict["tbit"] * tbit_conv  # Torque on the bit, kN*m
self.wob = temp_dict["wob"] * wob_conv  # Weight on bit, kN
self.rop = temp_dict["rop"] * rop_conv  # Rate of Penetration, m/h
self.an = temp_dict["an"] * an_conv   # Area of the nozzles, m^2
self.bit_n = temp_dict["bit_n"]  # drill bit efficiency
self.dp_e = temp_dict["dp_e"]  # drill pipe eccentricity

self.thao_o = temp_dict["thao_o"]
self.k = temp_dict["k"]
self.n = temp_dict["n"]

if temp_dict["visc"] == 0:
    n = self.n
    thao_w = ((self.q / (pi * n * (self.r3 - self.r2) ** 2 * (1 / (2 * (2 * n + 1) * self.k ** (1 / n))) *
            (self.r3 + self.r2))) + (self.thao_o * (2 * n + 1) / (n + 1)) ** (1 / n)) ** n
    shear_rate = ((thao_w - self.thao_o) / self.k) ** (1/n)
    self.visc_a = (self.thao_o / shear_rate) + self.k * shear_rate ** (n - 1)  # Fluid viscosity [Pas]

    if visc_eq:
        self.visc_p = self.visc_a
    else:
        from sympy import symbols, solve
        x = symbols('x')
        expr = self.q - (pi * n * self.r1 ** 3 * (1 / (3 * n + 1)) * (x / self.k) ** (1 / n) * (1 -
            (3 * n + 1) * (self.thao_o) / (n * (2 * n + 1) * x)))
        sol = solve(expr)
        thao_w_p = sol[0]
        shear_rate_p = ((thao_w_p - self.thao_o) / self.k) ** (1 / n)
        self.visc_p = float((self.thao_o / shear_rate_p) + self.k * shear_rate_p ** (n - 1))

else:
    self.visc_p = self.visc_a = temp_dict["visc"] / 1000

# HEAT COEFFICIENTS
if units == 'metric':
    lambda_conv = 1     # from W/(m*°C) to W/(m*°C)
    c_conv = 1  # from J/(kg*°C) to J/(kg*°C)
    gt_conv = 1     # from °C/m to °C/m
    beta_conv = 1   # from Pa to Pa
    alpha_conv = 1  # from 1/°F to 1/°C
else:
    lambda_conv = 1/1.73     # from BTU/(h*ft*°F) to W/(m*°C)
    c_conv = 4187.53  # from BTU/(lb*°F) to J/(kg*°C)
    gt_conv = 3.28*1.8     # from °F/ft to °C/m
    beta_conv = 6894.76  # from psi to Pa
    alpha_conv = 1.8  # from 1/°F to 1/°C
```

```
        # Thermal conductivity  W/(m*°C)
        self.lambdal = temp_dict["lambdal"] * lambda_conv  # Fluid
        self.lambdac = temp_dict["lambdac"] * lambda_conv   # Casing
        self.lambdacem = temp_dict["lambdacem"] * lambda_conv   # Cement
        self.lambdad = temp_dict["lambdad"] * lambda_conv   # Drill Pipe
        self.lambdafm = temp_dict["lambdafm"] * lambda_conv       # Formation
        self.lambdar = temp_dict["lambdar"] * lambda_conv      # Riser
        self.lambdaw = temp_dict["lambdaw"] * lambda_conv      # Seawater

        self.beta = temp_dict["beta"] * beta_conv  # isothermal bulk modulus, Pa
        self.alpha = temp_dict['alpha'] * alpha_conv     # Fluid Thermal Expansion Coefficient, 1/°C

        # Specific heat capacity, J/(kg*°C)
        self.cl = temp_dict["cl"] * c_conv       # Fluid
        self.cc = temp_dict["cc"] * c_conv     # Casing
        self.ccem = temp_dict["ccem"] * c_conv      # Cement
        self.cd = temp_dict["cd"] * c_conv      # Drill Pipe
        self.cr = temp_dict["cr"] * c_conv      # Riser
        self.cw = temp_dict["cw"] * c_conv       # Seawater
        self.cfm = temp_dict["cfm"] * c_conv       # Formation

        self.pr_p = self.visc_p * self.cl / self.lambdal       # Prandtl number
        self.pr_a = self.visc_a * self.cl / self.lambdal  # Prandtl number

        self.gt = temp_dict["gt"] * self.deltaz * gt_conv  # Geothermal gradient, from °C/m to °C/cell
        self.wtg = temp_dict["wtg"] * gt_conv * self.deltaz  # Seawater thermal gradient, from °C/m to °C/cell

        # Raise Errors:
        if self.casings[-1, 0] > self.dsro:
            raise ValueError('Last casing outer diameter must be smaller than the surrounding space diameter.')

        if self.casings[0, 2] > self.md[-1]:
            raise ValueError('MD must be higher than the first casing depth.')

        if self.casings[0, 1] < self.ddo:
            raise ValueError('Drill Pipe outer diameter must be smaller than the first casing inner diameter.')

        if self.wd > 0 and self.dro > self.dsro:
            raise ValueError('Riser diameter must be smaller than the surrounding space diameter.')

        if self.dsro > self.dfm:
            raise ValueError('Surrounding space diameter must be smaller than the undisturbed formation diameter.')

    def plot_torque_drag(self, plot='torque'):
        """
        Plot torque and drag forces
        :param plot: 'torque', 'drag' or 'both'
        :return: a plot
        """
        from .plot import plot_torque_drag
        plot_torque_drag(self, plot)

    def define_density(self, ic, cond=0):
        """
        Calculate the density profile
        :param ic: current temperature distribution
        :param cond: '0' to calculate the initial profile
        :return: density profile and derived calculations
        """

        from .fluid import initial_density, calc_density
        from .torque_drag import calc_torque_drag

        if cond == 0:
            self.rhof, self.rhof_initial = initial_density(self, ic)
        else:
            self.rhof = calc_density(self, ic, self.rhof_initial)
        self.drag, self.torque = calc_torque_drag(self)  # Torque/Forces, kN*m / kN
        self.re_p = [x * self.vp * 2 * self.r1 / self.visc_p for x in self.rhof]  # Reynolds number inside drill pipe
        self.re_a = [x * self.va * 2 * (self.r3 - self.r2) / self.visc_a for x in
                     self.rhof]  # Reynolds number - annular
        self.f_p = []  # Friction factor inside drill pipe
        self.nu_dpi = []
        self.nu_dpo = []
        for x in range(len(self.md)):
            if self.re_p[x] <= 2300:
                self.f_p.append(64 / self.re_p[x])
                self.nu_dpi.append(4.36)
                self.nu_dpo.append(4.36)

            if 2300 < self.re_p[x] < 10000:
                self.f_p.append(1.63 / log(6.9 / self.re_p[x]) ** 2)
                self.nu_dpi.append((self.f_p[x] / 8) * (self.re_p[x] - 1000) * self.pr_p /
                                   (1 + (12.7 * (self.f_p[x] / 8) ** 0.5) * (self.pr_p ** (2 / 3) - 1)))
                self.nu_dpo.append((self.f_p[x] / 8) * (self.re_a[x] - 1000) * self.pr_a /
                                   (1 + (12.7 * (self.f_p[x] / 8) ** 0.5) * (self.pr_a ** (2 / 3) - 1)))
            if self.re_p[x] >= 10000:
                self.f_p.append(1.63 / log(6.9 / self.re_p[x]) ** 2)
                self.nu_dpi.append(0.027 * (self.re_p[x] ** (4 / 5)) * (self.pr_p ** (1 / 3)) * (1 ** 0.14))
                self.nu_dpo.append(0.027 * (self.re_a[x] ** (4 / 5)) * (self.pr_a ** (1 / 3)) * (1 ** 0.14))

        self.h1 = [self.lambdal * x / self.ddi for x in self.nu_dpi]  # Drill Pipe inner wall
        self.h2 = [self.lambdal * x / self.ddo for x in self.nu_dpo]  # Drill Pipe outer wall
        self.nu_a = [1.86 * ((x * self.pr_a) ** (1 / 3)) * ((2 * (self.r3 - self.r2) / self.md[-1]) ** (1 / 3))
                     * (1 ** (1 / 4)) for x in self.re_a]
        # convective heat transfer coefficients, W/(m^2*°C)
        self.h3 = [self.lambdal * x / (2 * self.r3) for x in self.nu_a]  # Casing inner wall
        self.h3r = [self.lambdal * x / (2 * self.r3r) for x in self.nu_a]  # Riser inner wall

        return self
    return NewWell()
```

B.2. Initcond.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/drilling/initcond.py

```
def init_cond(well):
    """
    Generates the temperature profiles at time 0, before starting the operation.
    :param well: a well object created from the function set_well()
    :return: object with initial temperature profiles
    """

    # Initial Conditions
    Tdsio = [well.ts]   # Temperature of the fluid inside the drill string at RKB
    Tdso = [well.ts]    # Temperature of the drill string wall at RKB, t=0
    Tao = [well.ts]     # Temperature of the fluid inside the annulus at RKB, t=0
    Tcsgo = [well.ts]   # Temperature of the casing at RKB, t=0
    Tsro = [well.ts]    # Temperature of the surrounding space at RKB, t=0
    Tfm = [well.ts]     # Temperature of the formation at RKB

    for j in range(1, well.zstep):

        if j <= well.riser:
            Tg = well.wtg   # Water Thermal Gradient for the Riser section
        else:
            Tg = well.gt    # Geothermal Gradient below the Riser section

        deltaT = Tsro[j - 1] + Tg*(well.tvd[j]-well.tvd[j-1])/well.deltaz

        # Generating the Temperature Profile at t=0
        Tdsio.append(deltaT)
        Tdso.append(deltaT)
        Tao.append(deltaT)
        Tcsgo.append(deltaT)
        Tsro.append(deltaT)
        Tfm.append(deltaT)

    class InitCond(object):
        def __init__(self):
            self.tdsio = Tdsio
            self.tdso = Tdso
            self.tao = Tao
            self.tcsgo = Tcsgo
            self.tsro = Tsro
            self.tfm = Tfm

    return InitCond()
```

B.3. Heatcoefficients.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/drilling/heatcoefficients.py

```
def heat_coef(well, deltat):
    """
    Calculate heat transfer coefficients for each cell.
    :param well: a well object created from the function set_well()
    :param deltat: duration of each time step (seconds)
    :return: list with distribution of heat transfer coefficients
    """

    import math

    sections = [well.wd]
    if len(well.casings) > 0 and well.casings[0, 2] > 0:
        for i in range(len(well.casings))[::-1]:
            sections.append(well.casings[i, 2])

    # HEAT SOURCE TERMS

        # heat coefficients at bottom

    J = 4.1868    # Joule's constant  [Nm/cal]
    qbit = (1/J)*(1-well.bit_n)*(well.wob*(well.rop/3600)+2*math.pi*(well.rpm / 60)*well.tbit) \
        + 0.7 * (well.q/3600) * (well.rhof[-1]/(2*9.81)) * ((well.q/3600)/(0.95*well.an))**2

    vbit = well.q / well.an
    cbz = ((well.rhof[-1] * well.cl * vbit) / well.deltaz) / 2  # Vertical component (North-South)
    cbe = (2 * well.h1[-1] / well.r3) / 2  # East component
    cb = qbit / well.an  # Heat source term
    cbt = well.rhof[-1] * well.cl / deltat  # Time component

        # heat coefficients fluid inside annular

    qa = (0.085 * (2 * well.k * well.md[-1] / ((well.r3 - well.r2) * (127.094 * 10 ** 6))) *
         ((2 * (well.n + 1) * well.q) / (well.n * math.pi * (well.r3 + well.r2) *
           (well.r3 - well.r2) ** 2)) ** well.n) * (1 + (3/2) * well.dp_e**2)**0.5

    # Creating empty lists

    # Section 1: Fluid in Drill Pipe
    c1z = []
    c1e = []
    c1 = []
    c1t = []

    # Section 2: Drill Pipe Wall
    c2z = []
```

```
c2e = []
c2w = []
c2t = []

# Section 3: Fluid in Annulus
c3z = []
c3e = []
c3w = []
c3 = []
c3t = []

# Section 4: First casing
c4z = []
c4e = []
c4w = []
c4t = []

# Section 5: Surrounding Space
c5z = []
c5e = []
c5w = []
c5t = []

in_section = 1
section_checkpoint = sections[0]

for x in range(well.zstep):
    if x * well.deltaz >= section_checkpoint and in_section < len(sections) + 1:
        in_section += 1
        if section_checkpoint != sections[-1]:
            section_checkpoint = sections[in_section - 1]

    # heat coefficients fluid inside drill pipe

    qp = 2 * math.pi * (well.rpm / 60) * well.torque[1][x] \
        + 0.2 * well.q * 2 * (well.f_p[x] * well.rhof[x] * (well.vp ** 2) * (well.md[-1] /
                                                                              (well.ddi * 127.094 * 10 ** 6)))

    # fluid inside drill string
    c1z.append(((well.rhof[x] * well.cl * well.vp) / well.deltaz) / 2)  # Vertical component (North-South)
    c1e.append((2 * well.h1[x] / well.r1) / 2)  # East component
    c1.append(qp / (math.pi * (well.r1 ** 2)))  # Heat source term
    c1t.append(well.rhof[x] * well.cl / deltat)  # Time component

    # drill string wall
    c2z.append((well.lambdad / (well.deltaz ** 2)) / 2)  # Vertical component (North-South)
    c2e.append((2 * well.r2 * well.h2[x] / ((well.r2 ** 2) - (well.r1 ** 2))) / 2)  # East component
    c2w.append((2 * well.r1 * well.h1[x] / ((well.r2 ** 2) - (well.r1 ** 2))) / 2)  # West component
    c2t.append(well.rhod * well.cd / deltat)  # Time component
```

```
# fluid inside annular
c3z.append((well.rhof[x] * well.cl * well.va / well.deltaz) / 2)  # Vertical component (North-South)
c3e.append((2 * well.r3 * well.h3[x] / ((well.r3 ** 2) - (well.r2 ** 2))) / 2)  # East component
c3w.append((2 * well.r2 * well.h2[x] / ((well.r3 ** 2) - (well.r2 ** 2))) / 2)  # West component
c3.append(qa / (math.pi * ((well.r3 ** 2) - (well.r2 ** 2))))  # Heat source term
c3t.append(well.rhof[x] * well.cl / deltat)  # Time component

if in_section == 1:
  lambda4 = well.lambdar  # Thermal conductivity of the casing (riser in this section)
  lambda5 = well.lambdaw  # Thermal conductivity of the surrounding space (seawater)
  lambda45 = (lambda4 * (well.r4r - well.r3r) + lambda5 * (well.r5 - well.r4r)) / (
        well.r5 - well.r3r)  # Comprehensive Thermal conductivity of the casing (riser) and
                   # surrounding space (seawater)
  lambda56 = well.lambdaw  # Comprehensive Thermal conductivity of the surrounding space (seawater) and
                 # formation (seawater)
  c4 = well.cr  # Specific Heat Capacity of the casing (riser)
  c5 = well.cw  # Specific Heat Capacity of the surrounding space (seawater)
  rho4 = well.rhor  # Density of the casing (riser)
  rho5 = well.rhow  # Density of the surrounding space (seawater)

if 1 < in_section < len(sections):

  # calculation for surrounding space
  # thickness
  tcsr = 0
  tcem = 0
  for i in range(len(well.casings) - in_section):
    tcsr += (well.casings[i + 1, 0] - well.casings[i + 1, 1]) / 2
    tcem += (well.casings[i + 1, 1] - well.casings[i, 0]) / 2

    tcem += (well.casings[len(well.casings) - in_section + 1, 1] -
         well.casings[len(well.casings) - in_section, 0]) / 2
  if in_section == 2:
    tcem += (well.dsro - well.casings[-1, 0])
  xcsr = tcsr / (well.r5 - well.r4)  # fraction of surrounding space that is casing
  xcem = tcem / (well.r5 - well.r4)  # fraction of surrounding space that is cement
  xfm = 1 - xcsr - xcem  # fraction of surrounding space that is formation

  # thermal conductivity
  lambdasr = well.lambdac * xcsr + well.lambdacem * xcem + well.lambdafm * xfm
  lambdacsr = (well.lambdac * (well.r4 - well.r3) + lambdasr * (well.r5 - well.r4)) / (well.r5 - well.r3)
  lambdasrfm = (well.lambdac * (well.r5 - well.r4) + lambdasr * (well.rfm - well.r5)) / (well.rfm - well.r4)

  # Specific Heat Capacity
  csr = (well.cc * tcsr + well.ccem * tcem) / (well.r5 - well.r4)

  # Density
  rhosr = xcsr * well.rhoc + xcem * well.rhocem + xfm * well.rhofm

  lambda4 = well.lambdac
```

```
            lambda45 = lambdacsr
            lambda5 = lambdasr
            lambda56 = lambdasrfm
            c4 = well.cc  # Specific Heat Capacity of the casing
            c5 = csr  # Specific Heat Capacity of the surrounding space
            rho4 = well.rhoc  # Density of the casing
            rho5 = rhosr  # Density of the surrounding space

        if in_section == len(sections)+1:
            lambda4 = well.lambdafm
            lambda45 = well.lambdafm
            lambda5 = well.lambdafm
            lambda56 = well.lambdafm
            c4 = well.cfm  # Specific Heat Capacity of the casing (formation)
            c5 = well.cfm  # Specific Heat Capacity of the surrounding space (formation)
            rho4 = well.rhofm  # Density of the casing (formation)
            rho5 = well.rhofm  # Density of the surrounding space (formation)

        # first casing wall
        c4z.append((lambda4 / (well.deltaz ** 2)) / 2)
        c4e.append((2 * lambda45 / ((well.r4 ** 2) - (well.r3 ** 2))) / 2)
        c4w.append((2 * well.r3 * well.h3[x] / ((well.r4 ** 2) - (well.r3 ** 2))) / 2)
        c4t.append(rho4 * c4 / deltat)

        # surrounding space
        c5z.append((lambda5 / (well.deltaz ** 2)) / 2)
        c5w.append((lambda56 / (well.r5 * (well.r5 - well.r4) * math.log(well.r5 / well.r4))) / 2)
        c5e.append((lambda56 / (well.r5 * (well.r5 - well.r4) * math.log(well.rfm / well.r5))) / 2)
        c5t.append(rho5 * c5 / deltat)

    hc_1 = [c1z, c1e, c1, c1t]
    hc_2 = [c2z, c2e, c2w, c2t]
    hc_3 = [c3z, c3e, c3w, c3, c3t]
    hc_4 = [c4z, c4e, c4w, c4t]
    hc_5 = [c5z, c5e, c5w, c5t]
    coefficients = [hc_1, hc_2, hc_3, hc_4, hc_5, cb, cbe, cbt, cbz]

    return coefficients
```

B.4. Linearsystem.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/drilling/linearsystem.py

```
def define_coef(coefficients, zstep):
    """
    Retrieves respective heat transfer coefficients for certain depth point.
    :param coefficients: list with distribution of heat transfer coefficients
    :param zstep: depth step
    :return: values of heat coefficients for each section at the same depth
    """

    hc_1 = coefficients[0]
    c1z = hc_1[0][zstep]
    c1e = hc_1[1][zstep]
    c1 = hc_1[2][zstep]
    c1t = hc_1[3][zstep]

    hc_2 = coefficients[1]
    c2z = hc_2[0][zstep]
    c2e = hc_2[1][zstep]
    c2w = hc_2[2][zstep]
    c2t = hc_2[3][zstep]

    hc_3 = coefficients[2]
    c3z = hc_3[0][zstep]
    c3e = hc_3[1][zstep]
    c3w = hc_3[2][zstep]
    c3 = hc_3[3][zstep]
    c3t = hc_3[4][zstep]

    hc_4 = coefficients[3]
    c4z = hc_4[0][zstep]
    c4e = hc_4[1][zstep]
    c4w = hc_4[2][zstep]
    c4t = hc_4[3][zstep]

    hc_5 = coefficients[4]
    c5z = hc_5[0][zstep]
    c5e = hc_5[1][zstep]
    c5w = hc_5[2][zstep]
    c5t = hc_5[3][zstep]

    cb = coefficients[5]
    cbe = coefficients[6]
    cbt = coefficients[7]
    cbz = coefficients[8]

    return c1z, c1e, c1, c1t, c2z, c2e, c2w, c2t, c3z, c3e, c3w, c3, c3t, c4z, c4e, c4w, c4t, c5z, c5e, c5w, c5t, cb, \
        cbe, cbt, cbz


def temp_calc(well, initcond, heatcoeff):
    """
    Build the penta-diagonal matrix and solve it to get the well temperature distribution.
    :param well: a well object created from the function set_well()
    :param initcond: object with initial temperature profiles
    :param heatcoeff: list with distribution of heat transfer coefficients
    :return: object with final well temperature distribution
    """

    from numpy import zeros, linalg

    Tdsi = [well.tin]
    Tds = []
    Ta = []
    Tcsg = []
    Tsr = []
    xi = 5

    # Creating vectors N,W,C,E,S,B
    N = []
    W = []
    C = []
    E = []
    S = []
    B = []

    for j in range(well.zstep):
        c1z, c1e, c1, c1t, c2z, c2e, c2w, c2t, c3z, c3e, c3w, c3, c3t, c4z, c4e, c4w, c4t, c5z, c5e, c5w, c5t, cb, \
        cbe, cbt, cbz = define_coef(heatcoeff, j)

        for i in range(xi):
            if i == 0:  # Inside Drill String
                if j == 1:
                    W.append(0)
                    C.append(c1t + c1e + c1z)
                    E.append(-c1e)
                    S.append(0)
                    B.append(c1t * initcond.tdsio[j]    # Center(t=0)
                             + c1    # Heat Source
                             + c1e * (initcond.tdso[j] - initcond.tdsio[j])    # East(t=0)
                             + c1z * (initcond.tdsio[j - 1] - initcond.tdsio[j])    # N/S(t=0)
                             + c1z * (Tdsi[j - 1]))    # Tin

                if 1 < j < well.zstep - 1:
                    N.append(-c1z)
                    W.append(0)
                    C.append(c1t + c1e + c1z)
                    E.append(-c1e)
                    S.append(0)
                    B.append(c1t * initcond.tdsio[j]    # Center(t=0)
                             + c1       # Heat Source
                             + c1e * (initcond.tdso[j] - initcond.tdsio[j])    # East(t=0)
                             + c1z * (initcond.tdsio[j - 1] - initcond.tdsio[j]))    # N/S(t=0)

                if j == well.zstep - 1:     # Cell where fluid flows out of the tubing and then go to annular
                    N.append(-cbz)
                    W.append(0)
                    C.append(cbt + cbz + cbe)    # Note that c1t = c3t since it's the same fluid
                    E.append(-cbe)
                    B.append(cbt * initcond.tdsio[j]    # Center(t=0)
                             + cb       # Heat Source
                             + cbe * (initcond.tdso[j] - initcond.tdsio[j])    # East(t=0)
                             + cbz * (initcond.tdsio[j - 1] - initcond.tdsio[j]))    # N/S(t=0)

            if i == 1:  # Drill string wall

                if j == 0:
                    C.append(c2t + c2e + c2w + c2z)
                    E.append(-c2e)
                    S.append(-c2z)
                    B.append(c2t * initcond.tdso[j]
                             + c2w * Tdsi[j]
                             + c2e * (initcond.tao[j] - initcond.tdso[j])
                             + c2w * (initcond.tdsio[j] - initcond.tdso[j])
                             + c2z * (initcond.tdso[j + 1] - initcond.tdso[j]))

                if 0 < j < well.zstep - 1:
                    N.append(-c2z)
                    W.append(-c2w)
                    C.append(c2t + c2e + c2w + 2 * c2z)
                    E.append(-c2e)
                    if j < well.zstep - 2:
                        S.append(-c2z)
                    B.append(c2t * initcond.tdso[j]
                             + c2e * (initcond.tao[j] - initcond.tdso[j])
                             + c2w * (initcond.tdsio[j] - initcond.tdso[j])
                             + c2z * (initcond.tdso[j + 1] - initcond.tdso[j])
                             + c2z * (initcond.tdso[j - 1] - initcond.tdso[j]))

            if i == 2:  # Annular

                if j == 0:
                    W.append(-c3w)
                    C.append(c3t + c3e + c3w + c3z)
                    E.append(-c3e)
                    S.append(-c3z)
                    B.append(c3t * initcond.tao[j]
                        + c3
                        + c3e * (initcond.tcsgo[j] - initcond.tao[j])
                        + c3w * (initcond.tdso[j] - initcond.tao[j])
                        + c3z * (initcond.tao[j + 1] - initcond.tao[j]))

            if 0 < j < well.zstep - 1:
                N.append(0)
                W.append(-c3w)
                C.append(c3t + c3e + c3w + c3z)
                E.append(-c3e)
                if j < well.zstep - 2:
                    S.append(-c3z)
                B.append(c3t * initcond.tao[j]
                        + c3
                        + c3e * (initcond.tcsgo[j] - initcond.tao[j])
                        + c3w * (initcond.tdso[j] - initcond.tao[j])
                        + c3z * (initcond.tao[j + 1] - initcond.tao[j]))

if i == 3:  # Casing

    if j == 0:
        W.append(-c4w)
        C.append(c4t + c4e + c4w + c4z)
        E.append(- c4e)
        S.append(-c4z)
        B.append(c4t * initcond.tcsgo[j]   # Center(t=0)
                + c4e * (initcond.tsro[j] - initcond.tcsgo[j])  # East(t=0)
                + c4w * (initcond.tao[j] - initcond.tcsgo[j])  # West(t=0)
                + c4z * (initcond.tcsgo[j + 1] - initcond.tcsgo[j]))   # N/S(t=0)

    if 0 < j < well.zstep - 1:
        N.append(-c4z)
        W.append(-c4w)
        C.append(c4t + c4e + c4w + 2 * c4z)
        E.append(- c4e)
        S.append(-c4z)
        B.append(c4t * initcond.tcsgo[j]    # Center(t=0)
                + c4e * (initcond.tsro[j] - initcond.tcsgo[j])     # East(t=0)
                + c4w * (initcond.tao[j] - initcond.tcsgo[j])      # West(t=0)
                + c4z * (initcond.tcsgo[j + 1] - 2 * initcond.tcsgo[j] + initcond.tcsgo[j - 1])) # N/S(t=0)

    if j == well.zstep - 1:
        N.append(-c4z)
        W.append(-c4w)
        C.append(c4t + c4e + c4w + c4z)
        E.append(- c4e)
        B.append(c4t * initcond.tcsgo[j]    # Center(t=0)
                + c4e * (initcond.tsro[j] - initcond.tcsgo[j])     # East(t=0)
                + c4w * (initcond.tao[j] - initcond.tcsgo[j])      # West(t=0)
                + c4z * (initcond.tcsgo[j - 1] - initcond.tcsgo[j]))      # N/S(t=0)
    if i == 4:  # Surrounding Space

        if j == 0:
            W.append(-c5w)
            C.append(c5w + c5z + c5e + c5t)
            E.append(0)
            S.append(-c5z)
            B.append(c5w * (initcond.tcsgo[j] - initcond.tsro[j])
                     + c5z * (initcond.tsro[j + 1] - initcond.tsro[j])
                     + c5e * initcond.tsro[j]
                     + c5t * initcond.tsro[j])

        if 0 < j < well.zstep - 1:
            N.append(-c5z)
            W.append(-c5w)
            C.append(c5w + c5e + 2 * c5z + c5t)
            E.append(0)
            S.append(-c5z)
            B.append(c5w * (initcond.tcsgo[j] - initcond.tsro[j])
                     + c5z * (initcond.tsro[j + 1] - initcond.tsro[j])
                     + c5z * (initcond.tsro[j - 1] - initcond.tsro[j])
                     + c5e * initcond.tsro[j] +
                     c5t * initcond.tsro[j])

        if j == well.zstep - 1:
            N.append(-c5z)
            W.append(-c5w)
            C.append(c5w + c5e + c5z + c5t)
            B.append(c5w * (initcond.tcsgo[j] - initcond.tsro[j])
                     + c5z * (initcond.tsro[j - 1] - initcond.tsro[j])
                     + c5e * initcond.tsro[j]
                     + c5t * initcond.tsro[j])

#LINEARSYSTEM
# Creating pentadiagonal matrix
A = zeros((xi * well.zstep - 3, xi * well.zstep - 3))

# Filling up Pentadiagonal Matrix A
lenC = xi * well.zstep - 3
lenN = lenC - xi
lenW = lenC - 1
lenE = lenC - 1
lenS = lenC - xi

for it in range(lenC):  # Inserting list C
    A[it, it] = C[it]
for it in range(lenE):  # Inserting list E
    A[it, it + 1] = E[it]
for it in range(lenW):  # Inserting list W
```

```
    A[it + 1, it] = W[it]
for it in range(lenN):  # Inserting list N
    A[it + xi, it] = N[it]
for it in range(lenS):  # Inserting list S
    A[it, it + xi] = S[it]

A[lenC - 1 - (xi - 3) - (xi - 1), lenC - 1 - (xi - 3)] = -c2z
A[lenC - 1 - (xi - 3) - (xi - 2), lenC - 1 - (xi - 3)] = -c3z

Temp = linalg.solve(A, B)

for x in range(well.zstep):
    if x < well.zstep - 1:
        Tds.append(Temp[5 * x])
    if x == well.zstep - 1:
        Tds.append(Temp[lenC - 1 - (xi - 3)])
for x in range(well.zstep - 1):
    if x < well.zstep - 2:
        Tdsi.append(Temp[5 * x + 4])
    if x == well.zstep - 2:
        Tdsi.append(Temp[lenC - 1 - (xi - 3)])
for x in range(well.zstep):
    if x < well.zstep - 1:
        Ta.append(Temp[5 * x + 1])
    if x == well.zstep - 1:
        Ta.append(Temp[lenC - 1 - (xi - 3)])
for x in range(well.zstep):
    if x < well.zstep - 1:
        Tcsg.append(Temp[5 * x + 2])
    if x == well.zstep - 1:
        Tcsg.append(Temp[lenC - 2])
for x in range(well.zstep):
    if x < well.zstep - 1:
        Tsr.append(Temp[5 * x + 3])
    if x == well.zstep - 1:
        Tsr.append(Temp[lenC - 1])

t3 = Tcsg.copy()

Tr = Tcsg[:well.riser]+[None]*(well.zstep-well.riser)
for x in range(well.riser):
    Tcsg[x] = None

csgs_reach = int(well.casings[0, 2] / well.deltaz)    # final depth still covered with casing(s)

Toh = [None]*csgs_reach + Tcsg[csgs_reach:]
for x in range(csgs_reach, well.zstep):
    Tcsg[x] = None

class TempCalc(object):
        def __init__(self):
            self.tdsi = Tdsi
            self.tds = Tds
            self.ta = Ta
            self.tr = Tr
            self.t3 = t3
            self.tcsg = Tcsg
            self.toh = Toh
            self.tsr = Tsr
            self.csgs_reach = csgs_reach

    return TempCalc()
```

B.5. Torque_drag.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/drilling/torque_drag.py

```
def calc_torque_drag(well):
    """
    Function to generate the torque and drag profiles. Model Source: SPE-11380-PA
    :param well: a well object with rhod (drill string density), r1 (inner diameter of drill string), r2 (outer diameter
    of drill string), r3 (diameter of the first casing layer or borehole), rhof (fluid density), deltaz (length per pipe
    segment), wob (weight on bit), tbit (torque on bit), azimuth (for each segment) and inclination (for each segment).
    :return: two lists, drag force and torque
    """

    from math import pi, sin, cos, radians

    fric = 0.24     # sliding friction coefficient between DP-wellbore
    unit_pipe_weight = well.rhod * 9.81 * pi * (well.r2 ** 2 - well.r1 ** 2)
    area_a = pi * ((well.r3 ** 2) - (well.r2 ** 2))
    area_ds = pi * (well.r1 ** 2)
    buoyancy = [1 - ((x * area_a) - (x * area_ds)) / (well.rhod * (area_a - area_ds)) for x in well.rhof]
    w = [unit_pipe_weight * well.deltaz * x for x in buoyancy]
    w[0] = 0

    force_1, force_2, force_3 = [well.wob], [well.wob], [well.wob]      # Force at bottom
    torque_1, torque_2, torque_3 = [well.tbit], [well.tbit], [well.tbit]      # Torque at bottom

    for x in reversed(range(1, well.zstep)):
        delta_azi = radians(well.azimuth[x] - well.azimuth[x-1])
        delta_inc = radians(well.inclination[x] - well.inclination[x-1])
        inc_avg = radians((well.inclination[x] + well.inclination[x-1]) / 2)

        # NET NORMAL FORCES
        fn_1 = ((force_1[-1] * delta_azi * sin(inc_avg)) ** 2
                + (force_1[-1] * delta_inc + w[x] * sin(inc_avg)) ** 2) ** 0.5
        fn_2 = ((force_2[-1] * delta_azi * sin(inc_avg)) ** 2
                + (force_2[-1] * delta_inc + w[x] * sin(inc_avg)) ** 2) ** 0.5
        fn_3 = ((force_3[-1] * delta_azi * sin(inc_avg)) ** 2
                + (force_3[-1] * delta_inc + w[x] * sin(inc_avg)) ** 2) ** 0.5

        # DRAG FORCE CALCULATIONS
        delta_ft_1 = w[x] * cos(inc_avg) - fric * fn_1
        delta_ft_2 = w[x] * cos(inc_avg)
        delta_ft_3 = w[x] * cos(inc_avg) + fric * fn_3

        ft_1 = force_1[-1] + delta_ft_1
        ft_2 = force_2[-1] + delta_ft_2
        ft_3 = force_3[-1] + delta_ft_3

        force_1.append(ft_1)
        force_2.append(ft_2)
        force_3.append(ft_3)

        # TORQUE CALCULATIONS
        delta_torque_1 = fric * fn_1 * well.r2
        delta_torque_2 = fric * fn_2 * well.r2
        delta_torque_3 = fric * fn_3 * well.r2

        t_1 = torque_1[-1] + delta_torque_1
        t_2 = torque_2[-1] + delta_torque_2
        t_3 = torque_3[-1] + delta_torque_3

        torque_1.append(t_1)
        torque_2.append(t_2)
        torque_3.append(t_3)

    # Units conversion: N to kN
    force = [[i/1000 for i in force_1[::-1]], [i/1000 for i in force_2[::-1]], [i/1000 for i in force_3[::-1]]]
    torque = [[i/1000 for i in torque_1[::-1]], [i/1000 for i in torque_2[::-1]], [i/1000 for i in torque_3[::-1]]]

    return force, torque
```

B.6. Fluid.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/drilling/fluid.py

```
def initial_density(well, initcond):
    """
    Function to calculate the density profile for the first time step
    :param well: a well object created from the function set_well()
    :param initcond: a initial conditions object with the formation temperature profile
    :return: the density profile and the initial density at surface conditions
    """

    rhof_initial = well.rhof
    pressure = [well.rhof * 9.81 * i for i in well.tvd]
    rhof = [well.rhof * (1 + (x - 10 ** 5) / well.beta - well.alpha * (y - well.ts)) for x, y in
            zip(pressure, initcond.tdsio)]
    pressure = [x * 9.81 * y for x, y in zip(rhof, well.tvd)]
    rhof = [well.rhof * (1 + (x - 10 ** 5) / well.beta - well.alpha * (y - well.ts)) for x, y in
            zip(pressure, initcond.tdsio)]

    return rhof, rhof_initial


def calc_density(well, initcond, rhof_initial):
    """
    Function to calculate the density profile
    :param well: a well object created from the function set_well()
    :param initcond: a initial conditions object with the formation temperature profile
    :param rhof_initial: initial density at surface conditions
    :return: density profile
    """

    pressure_h = [x * 9.81 * y for x, y in zip(well.rhof, well.tvd)]
    pressure_f = [x * (well.md[-1] / well.ddi) * (1/2) * y * well.vp **2 for x, y in zip(well.f_p, well.rhof)]
    pressure = [x + y for x, y in zip(pressure_h, pressure_f)]
    rhof = [rhof_initial * (1 + (x - 10 ** 5) / well.beta - well.alpha * (y - well.ts)) for x, y in
            zip(pressure, initcond.tdsio)]

    return rhof
```

B.7. Main.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/drilling/main.py

```
def temp_time(n, well, log=True, units='metric', density_constant=False, time_delta=None):
    """
    Function to calculate the well temperature distribution during drilling at a certain circulation time n.
    :param n: circulation time, hours
    :param well: a well object created from the function set_well()
    :param log: save distributions between initial time and circulation time n (each 1 hour)
    :param units: system of units ('metric' or 'english')
    :param density_constant: keep the fluid density as a constant
    :param time_delta: duration of each time step (seconds)
    :return: a temperature distribution object
    """
    from .initcond import init_cond
    from .heatcoefficients import heat_coef
    from .linearsystem import temp_calc
    from .plot import profile
    from math import log, nan
    import numpy as np
    # Simulation main parameters
    time = n  # circulating time, h
    tcirc = time * 3600  # circulating time, s
    deltat = 60 * time
    if type(time_delta) == int:
        deltat = time_delta
    tstep = int(tcirc / deltat)
    ic = init_cond(well)
    tfm = ic.tfm
    well = well.define_density(ic, cond=0)
    if density_constant:
        deltat = tcirc
        tstep = 1
    hc = heat_coef(well, deltat)
    temp = temp_calc(well, ic, hc)

    if not density_constant:
        temp.tdsi = temp.tds = temp.ta = temp.t3 = temp.tsr = tfm
        for x in range(len(tfm)):
            if temp.tcsg[x] != nan:
                temp.tcsg[x] = tfm[x]
            if temp.tr[x] != nan:
                temp.tr[x] = tfm[x]
            if temp.toh[x] != nan:
                temp.toh[x] = tfm[x]

    temp_initial = temp
    temp_initial.tdsi = ic.tfm
    temp_initial.tds = ic.tfm
    temp_initial.ta = ic.tfm

    temp_log = [temp_initial, temp]
    time_log = [0, deltat / 3600]

    for x in range(tstep-1):
        if tstep > 1:
            well = well.define_density(ic, cond=1)
            ic.tdsio = temp.tdsi
            ic.tdso = temp.tds
            ic.tao = temp.ta
            ic.tcsgo = temp.t3
            ic.tsr = temp.tsr
            hc_new = heat_coef(well, deltat)
            temp = temp_calc(well, ic, hc_new)

        if units == 'english':
            temp.tdsi_output = [(i/(5/9)+32) for i in temp.tdsi]
            temp.tds_output = [(i/(5/9)+32) for i in temp.tds]
            temp.ta_output = [(i/(5/9)+32) for i in temp.ta]
            temp.tcsg_output = [(i/(5/9)+32) for i in temp.tcsg if type(i) == np.float64]
            temp.tr_output = [(i/(5/9)+32) for i in temp.tr if type(i) == np.float64]
            temp.tsr_output = [(i/(5/9)+32) for i in temp.tsr]
            temp.md_output = [i*3.28 for i in well.md]

        if log:
            temp_log.append(temp)
            time_log.append(time_log[-1] + time_log[1])

    if units == 'english':
        temp.tdsi = temp.tdsi_output
        temp.tds = temp.tds_output
        temp.ta = temp.ta_output
        temp.tcsg = temp.tcsg_output
        temp.tr = temp.tr_output
        temp.sr = temp.tsr_output
        temp.md = temp.md_output
        tfm = [(i / (5 / 9) + 32) for i in tfm]

    class TempDist(object):
        def __init__(self):
            self.tdsi = temp.tdsi
            self.tds = temp.tds
            self.ta = temp.ta
            self.tr = temp.tr
            self.tcsg = temp.tcsg
            self.toh = temp.toh
            self.tsr = temp.tsr
            self.tfm = tfm
```

```
            self.time = time
            self.md = well.md
            self.riser = well.riser
            self.csgs_reach = temp.csgs_reach
            self.deltat = deltat
            if log:
                self.temp_log = temp_log
                self.time_log = time_log

        def plot(self, tdsi=True, ta=True, tr=False, tcsg=False, tfm=True, sr=False):
            profile(self, tdsi, ta, tr, tcsg, tfm, sr, units)

        def well(self):
            return well

        def behavior(self):
            temp_behavior_drilling = temp_behavior(self)
            return temp_behavior_drilling

        def plot_multi(self, tdsi=True, ta=False, tr=False, tcsg=False, tfm=False, tsr=False):
            plot_multitime(self, tdsi, ta, tr, tcsg, tfm, tsr)

    return TempDist()


def temp_behavior(temp_dist):

    ta = [x.ta for x in temp_dist.temp_log]
    tbot = []
    tout = []

    for n in range(len(ta)):
        tbot.append(ta[n][-1])
        tout.append(ta[n][0])

    class Behavior(object):
        def __init__(self):
            self.finaltime = temp_dist.time
            self.tbot = tbot
            self.tout = tout
            self.tfm = temp_dist.tfm
            self.time = temp_dist.time_log

        def plot(self):
            from .plot import behavior
            behavior(self)

    return Behavior()
```

```
def plot_multitime(temp_dist, tdsi=True, ta=False, tr=False, tcsg=False, tfm=False, tsr=False):
    from .plot import profile_multitime

    values = temp_dist.temp_log
    times = [x for x in temp_dist.time_log]
    profile_multitime(temp_dist, values, times, tdsi=tdsi, ta=ta, tr=tr, tcsg=tcsg, tfm=tfm, tsr=tsr)


# BUILDING GENERAL FUNCTIONS FOR DRILLING MODULE


def temp(n, mdt=3000, casings=[], wellpath_data=[], d_openhole=0.216, grid_length=50, profile='V', build_angle=1,
kop=0,
         eob=0, sod=0, eod=0, kop2=0, eob2=0, change_input={}, log=False, visc_eq=True, units='metric',
         density_constant=False, time_delta=None):
    """
    Main function to calculate the well temperature distribution during drilling operation. This function allows to
    set the wellpath and different parameters involved.
    :param n: circulation time, hours
    :param mdt: measured depth of target, m
    :param casings: list of dictionaries with casings characteristics (od, id and depth)
    :param wellpath_data: load own wellpath as a list
    :param d_openhole: diameter of open hole section, m
    :param grid_length: number of cells through depth
    :param profile: type of well to generate. Vertical ('V'), S-type ('S'), J-type ('J') and Horizontal ('H1' or 'H2')
    :param build_angle: build angle, °
    :param kop: kick-off point, m
    :param eob: end of build, m
    :param sod: start of drop, m
    :param eod: end of drop, m
    :param kop2: kick-off point 2, m
    :param eob2: end of build 2, m
    :param change_input: dictionary with parameters to set.
    :param log: save distributions between initial time and circulation time n (each 1 hour)
    :param visc_eq: boolean to use the same viscosity in the pipe and annular
    :param units: system of units ('metric' or 'english')
    :param density_constant: keep the fluid density as a constant
    :param time_delta: duration of each time step (seconds)
    :return: a well temperature distribution object
    """
    from .input import data, set_well
    from .. import wellpath

    tdata = data(casings, d_openhole, units)

    for x in change_input:   # changing default values
        if x in tdata:
            tdata[x] = change_input[x]
        else:
            raise TypeError('%s is not a parameter' % x)
    if len(wellpath_data) == 0:
        depths = wellpath.get(mdt, grid_length, profile, build_angle, kop, eob, sod, eod, kop2, eob2, units)
    else:
        depths = wellpath.load(wellpath_data, grid_length, units)
    well = set_well(tdata, depths, visc_eq, units)
    temp_distribution = temp_time(n, well, log, units, density_constant, time_delta)

    return temp_distribution


def input_info(about='all'):
    from .input import info
    info(about)
```

Appendix C Production Module

C.1. Input.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/production/input.py

```
def data(casings=[], d_openhole=0.216, units='metric'):
    """
    Parameters involved within the operation calculations
    :param casings: list of dictionaries with casings characteristics (od, id and depth)
    :param d_openhole: diameter of open hole section, m
    :param units: system of units ('metric' or 'english')
    :return: a dictionary with default values for the required parameters
    """

    from numpy import asarray

    dict_met = {'ts': 15.0, 'wd': 100.0,  'dti': 4.0, 'dto': 4.5, 'dri': 17.716, 'dro': 21.0, 'dfm': 80.0,
        'q': 2000, 'lambdaf': 0.635, 'lambdac': 43.3, 'lambdacem': 0.7, 'lambdat': 40.0, 'lambdafm': 2.249,
        'lambdar': 15.49, 'lambdaw': 0.6, 'cf': 3713.0, 'cc': 469.0, 'ccem': 2000.0, 'ct': 400.0, 'cr': 464.0,
        'cw': 4000.0, 'cfm': 800.0, 'rhof': 0.85, 'rhof_a': 1.2, 'rhot': 7.8, 'rhoc': 7.8, 'rhor': 7.8,
        'rhofm': 2.245, 'rhow': 1.029, 'rhocem': 2.7, 'gt': 0.0238, 'wtg': -0.005, 'visc': 15,
        'beta': 44983 * 10 ** 5, 'alpha': 960 * 10 ** -6, 'beta_a': 44983 * 10 ** 5, 'alpha_a': 960 * 10 ** -6}

    dict_eng = {'ts': 59.0, 'wd': 328.0, 'dti': 4.0, 'dto': 4.5, 'dri': 17.716, 'dro': 21.0, 'dfm': 80.0,
          'q': 366.91, 'lambdaf': 1.098, 'lambdac': 74.909, 'lambdacem': 1.21, 'lambdat': 69.2, 'lambdafm': 3.89,
          'lambdar': 26.8, 'lambdaw': 1.038, 'cf': 0.887, 'cc': 0.112, 'ccem': 0.478, 'ct': 0.096, 'cr': 0.1108,
          'cw': 0.955, 'cfm': 0.19, 'rhof': 7.09, 'rhof_a': 10, 'rhot': 65.09, 'rhoc': 65.09, 'rhor': 65.09,
          'rhofm': 18.73, 'rhow': 8.587, 'rhocem': 22.5, 'gt': 0.00403, 'wtg': -8.47*10**-4, 'visc': 15,
          'beta': 652423, 'alpha': 5.33 * 10 ** -4, 'beta_a': 652423, 'alpha_a': 5.33 * 10 ** -4}

    if units == 'metric':
        dict = dict_met
    else:
        dict = dict_eng

    if len(casings) > 0:
        od = sorted([x['od'] * 0.0254 for x in casings])
        id = sorted([x['id'] * 0.0254 for x in casings])
        depth = sorted([x['depth'] for x in casings], reverse=True)
        dict['casings'] = [[od[x], id[x], depth[x]] for x in range(len(casings))]
        dict['casings'] = asarray(dict['casings'])
    else:
        dict['casings'] = [[(d_openhole + dict['dro'] * 0.0254), d_openhole, 0]]
        dict['casings'] = asarray(dict['casings'])

    return dict
```

```
def info(about='all'):
    """
    Retrieves information about the parameters (description and units)
    :param about: type of parameters
    :return: description and units of parameters
    """

    print("Use the ID of a parameter to change the default value (e.g. tdict['tin']=30 to change the fluid inlet "
          "temperature from the default value to 30° Celsius)")
    print('Notice that the information is provided as follows:' + '\n' +
          'parameter ID: general description, units' + '\n')

    tubular_parameters = 'VALUES RELATED TO TUBULAR SIZES' + '\n' + \
                         'dti: tubing inner diameter, in' + '\n' + \
                         'dto: tubing outer diameter, in' + '\n' + \
                         'dri: riser inner diameter, in' + '\n' + \
                         'dro: riser outer diameter, in' + '\n'

    conditions_parameters = 'PARAMETERS RELATED TO SIMULATION CONDITIONS' + '\n' + \
                            'ts: surface temperature, °C or °F' + '\n' + \
                            'wd: water depth, m or ft' + '\n' + \
                            'dfm: undisturbed formation diameter, m or ft' + '\n'

    heatcoeff_parameters = 'PARAMETERS RELATED TO HEAT COEFFICIENTS' + '\n' + \
                           'lambdaf: fluid - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdac: casing - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdacem: cement - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdat: tubing - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdafm: formation - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdar: riser - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdaw: water - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'cf: fluid - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'cc: casing - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'ccem: cement - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'ct: tubing - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'cr: riser - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'cw: water - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'cfm: formation - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'gt: geothermal gradient, °C/m or °F/ft' + '\n' + \
                           'wtg: seawater thermal gradient, °C/m or °F/ft' + '\n'

    densities_parameters = 'PARAMETERS RELATED TO DENSITIES' + '\n' + \
                           'rhof: fluid density, sg or ppg' + '\n' + \
                           'rhot: tubing density, sg or ppg' + '\n' + \
                           'rhoc: casing density, sg or ppg' + '\n' + \
                           'rhor: riser density, sg or ppg' + '\n' + \
                           'rhofm: formation density, sg or ppg' + '\n' + \
                           'rhow: seawater density, sg or ppg' + '\n' + \
                           'rhocem: cement density, sg or ppg' + '\n' + \
                           'beta: isothermal bulk modulus of production fluid, Pa' + '\n' + \
                           'alpha: expansion coefficient of production fluid, 1/°C' + '\n' + \
                           'beta_a: isothermal bulk modulus of fluid in annular, Pa' + '\n' + \
                           'alpha_a: expansion coefficient of fluid in annular, 1/°C or 1/°F' + '\n'

    viscosity_parameters = 'PARAMETERS RELATED TO MUD VISCOSITY' + '\n' + \
                           'thao_o: yield stress, Pa or psi' + '\n' + \
                           'n: flow behavior index, dimensionless' + '\n' + \
                           'k: consistency index, Pa*s^n or psi*s^n' + '\n' + \
                           'visc: fluid viscosity, cp' + '\n'

    operational_parameters = 'PARAMETERS RELATED TO THE OPERATION' + '\n' + \
                       'q: flow rate, m^3/day or bbl/day' + '\n'

    if about == 'casings':
        print(tubular_parameters)

    if about == 'conditions':
        print(conditions_parameters)

    if about == 'heatcoeff':
        print(heatcoeff_parameters)

    if about == 'densities':
        print(densities_parameters)

    if about == 'operational':
        print(operational_parameters)

    if about == 'viscosity':
        print(viscosity_parameters)

    if about == 'all':
        print(tubular_parameters + '\n' + conditions_parameters + '\n' + heatcoeff_parameters + '\n' +
              densities_parameters + '\n' + viscosity_parameters + '\n' + operational_parameters)


def set_well(temp_dict, depths, units='metric'):
    """
    Define properly the parameters and respective values within an object well.
    :param temp_dict: dictionary with inputs and default values.
    :param depths: wellpath object
    :param units: system of units ('metric' or 'english')
    :return: a well object with conditions and parameters defined
    """

    from math import pi, log

    def wellpath():
        """
        :return: wellpath object
        """
        return depths

class NewWell(object):
    def __init__(self):
        # DEPTH
        self.md = depths.md
        self.tvd = depths.tvd
        self.deltaz = depths.deltaz
        self.zstep = depths.zstep
        self.sections = depths.sections
        self.north = depths.north
        self.east = depths.east
        self.inclination = depths.inclination
        self.dogleg = depths.dogleg
        self.azimuth = depths.azimuth
        if units != 'metric':
            self.md = [i / 3.28 for i in self.md]
            self.tvd = [i / 3.28 for i in self.tvd]
            self.deltaz = self.deltaz / 3.28
            self.north = [i / 3.28 for i in self.north]
            self.east = [i / 3.28 for i in self.east]

        # TUBULAR
        if units == 'metric':
            d_conv = 0.0254   # from in to m
        else:
            d_conv = 0.0254   # from in to m
        self.casings = temp_dict["casings"]  # casings array
        self.dti = temp_dict["dti"] * d_conv  # Tubing Inner  Diameter, m
        self.dto = temp_dict["dto"] * d_conv   # Tubing Outer Diameter, m
        self.dri = temp_dict["dri"] * d_conv  # Riser diameter Inner Diameter, m
        self.dro = temp_dict["dro"] * d_conv   # Riser diameter Outer Diameter, m

        # CONDITIONS
        if units == 'metric':
            depth_conv = 1  # from m to m
            self.ts = temp_dict["ts"]  # Surface Temperature (RKB), °C
        else:
            depth_conv = 1 / 3.28  # from ft to m
            self.ts = (temp_dict["ts"] - 32) * (5 / 9)  # Surface Temperature (RKB), from °F to °C
        self.wd = temp_dict["wd"] * depth_conv  # Water Depth, m
        self.riser = round(self.wd / self.deltaz)  # number of grid cells for the riser
        self.dsr = self.casings[0, 0]  # Surrounding Space Inner Diameter, m
        self.dsro = sorted([self.dro + 0.03, self.casings[-1, 0] + 0.03])[-1]  # Surrounding Space Outer Diameter, m
        self.dfm = temp_dict["dfm"] * d_conv  # Undisturbed Formation Diameter, m

        # RADIUS (CALCULATED)
```

```
self.r1 = self.dti / 2  # Tubing Inner  Radius, m
self.r2 = self.dto / 2  # Tubing Outer Radius, m
self.r3 = self.casings[0, 1] / 2  # Casing Inner Radius, m
self.r3r = self.dri / 2  # Riser Inner Radius, m
self.r4r = self.dro / 2  # Riser Outer Radius, m
self.r4 = self.casings[0, 0] / 2  # Surrounding Space Inner Radius m
self.r5 = self.dsro / 2  # Surrounding Space Outer Radius, m
self.rfm = self.dfm / 2  # Undisturbed Formation Radius, m

# DENSITIES kg/m3
if units == 'metric':
    dens_conv = 1000     # from sg to kg/m3
else:
    dens_conv = 119.83   # from ppg to kg/m3
self.rhof = temp_dict["rhof"] * dens_conv  # Fluid
self.rhof_a = temp_dict["rhof_a"] * dens_conv  # Fluid
self.rhot = temp_dict["rhot"] * dens_conv  # Tubing
self.rhoc = temp_dict["rhoc"] * dens_conv  # Casing
self.rhor = temp_dict["rhor"] * dens_conv  # Riser
self.rhocem = temp_dict["rhocem"] * dens_conv  # Cement Sheath
self.rhofm = temp_dict["rhofm"] * dens_conv  # Formation
self.rhow = temp_dict["rhow"] * dens_conv  # Seawater
self.visc = temp_dict["visc"] / 1000  # Fluid viscosity [Pas]

# OPERATIONAL
if units == 'metric':
    q_conv = 0.04167     # from m^3/day to m^3/h
else:
    q_conv = 0.0066244706   # from bbl/day to m^3/h
self.q = temp_dict["q"] * q_conv  # Flow rate, m^3/h
self.vp = (self.q / (pi * (self.r1 ** 2))) / 3600  # Fluid velocity through the tubing

# HEAT COEFFICIENTS
if units == 'metric':
    lambda_conv = 1     # from W/(m*°C) to W/(m*°C)
    c_conv = 1  # from J/(kg*°C) to J/(kg*°C)
    gt_conv = 1     # from °C/m to °C/m
    beta_conv = 1   # from Pa to Pa
    alpha_conv = 1  # from 1/°F to 1/°C
else:
    lambda_conv = 1/1.73     # from BTU/(h*ft*°F) to W/(m*°C)
    c_conv = 4187.53  # from BTU/(lb*°F) to J/(kg*°C)
    gt_conv = 3.28*1.8     # from °F/ft to °C/m
    beta_conv = 6894.76  # from psi to Pa
    alpha_conv = 1.8  # from 1/°F to 1/°C

# Thermal conductivity  W/(m*°C)
self.lambdaf = temp_dict["lambdaf"] * lambda_conv  # Fluid
self.lambdac = temp_dict["lambdac"] * lambda_conv   # Casing
self.lambdacem = temp_dict["lambdacem"] * lambda_conv   # Cement
```

```
        self.lambdat = temp_dict["lambdat"] * lambda_conv   # Tubing wall
        self.lambdafm = temp_dict["lambdafm"] * lambda_conv       # Formation
        self.lambdar = temp_dict["lambdar"] * lambda_conv      # Riser
        self.lambdaw = temp_dict["lambdaw"] * lambda_conv      # Seawater

        self.beta = temp_dict["beta"] * beta_conv      # isothermal bulk modulus in tubing, Pa
        self.alpha = temp_dict['alpha'] * alpha_conv    # Fluid Thermal Expansion Coefficient in tubing, 1/°C
        self.beta_a = temp_dict["beta_a"] * beta_conv       # isothermal bulk modulus in annular, Pa
        self.alpha_a = temp_dict['alpha_a'] * alpha_conv    # Fluid Thermal Expansion Coefficient in annular, 1/°C

        # Specific heat capacity, J/(kg*°C)
        self.cf = temp_dict["cf"] * c_conv      # Fluid
        self.cc = temp_dict["cc"] * c_conv    # Casing
        self.ccem = temp_dict["ccem"] * c_conv    # Cement
        self.ct = temp_dict["ct"] * c_conv    # Tubing
        self.cr = temp_dict["cr"] * c_conv    # Riser
        self.cw = temp_dict["cw"] * c_conv     # Seawater
        self.cfm = temp_dict["cfm"] * c_conv      # Formation

        self.pr = self.visc * self.cf / self.lambdaf      # Prandtl number

        self.gt = temp_dict["gt"] * gt_conv * self.deltaz  # Geothermal gradient, °C/m
        self.wtg = temp_dict["wtg"] * gt_conv * self.deltaz  # Seawater thermal gradient, °C/m

        # Raise Errors:

        if self.casings[-1, 0] > self.dsro:
            raise ValueError('Last casing outer diameter must be smaller than the surrounding space diameter.')

        if self.casings[0, 2] > self.md[-1]:
            raise ValueError('MD must be higher than the first casing depth.')

        if self.casings[0, 1] < self.dto:
            raise ValueError('Tubing outer diameter must be smaller than the first casing inner diameter.')

        if self.wd > 0 and self.dro > self.dsro:
            raise ValueError('Riser diameter must be smaller than the surrounding space diameter.')

        if self.dsro > self.dfm:
            raise ValueError('Surrounding space diameter must be smaller than the undisturbed formation diameter.')

    def define_density(self, ic, cond=0):
        """
        Calculate the density profile
        :param ic: current temperature distribution
        :param cond: '0' to calculate the initial profile
        :return: density profile and derived calculations
        """

        from .fluid import initial_density, calc_density
        if cond == 0:
            self.rhof, self.rhof_initial = initial_density(self, ic)
            self.rhof_a, self.rhof_a_initial = initial_density(self, ic, section='annular')
        else:
            self.rhof = calc_density(self, ic, self.rhof_initial)
            self.rhof_a = calc_density(self, ic, self.rhof_initial, section='annular')
        self.re_p = [x * self.vp * 2 * self.r1 / self.visc for x in self.rhof]  # Reynolds number inside tubing
        self.f_p = []  # Friction factor inside tubing
        self.nu_dpi = []
        for x in range(len(self.md)):
            if self.re_p[x] < 2300:
                self.f_p.append(64 / self.re_p[x])
                self.nu_dpi.append(4.36)
            else:
                self.f_p.append(1.63 / log(6.9 / self.re_p[x]) ** 2)
                self.nu_dpi.append(
                    (self.f_p[x] / 8) * (self.re_p[x] - 1000) * self.pr / (1 + (12.7 * (self.f_p[x] / 8) ** 0.5) *
                                                                (self.pr ** (2 / 3) - 1)))
        # convective heat transfer coefficients, W/(m^2*°C)
        self.h1 = [self.lambdaf * x / self.dti for x in self.nu_dpi]  # Tubing inner wall
        return self

    return NewWell()
```

C.2. Initcond.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/production/initcond.py

```
def init_cond(well):
    """
    Generates the temperature profiles at time 0, before starting the operation.
    :param well: a well object created from the function set_well()
    :return: object with initial temperature profiles
    """

    # Initial Conditions
    Tfto = [well.ts]   # Temperature of the fluid inside the tubing at RKB
    Tto = [well.ts]    # Temperature of the tubing at RKB, t=0
    Tao = [well.ts]      # Temperature of the fluid inside the annulus at RKB, t=0
    Tco = [well.ts]    # Temperature of the casing at RKB, t=0
    Tsro = [well.ts]    # Temperature of the surrounding space at RKB, t=0
    Tfm = [well.ts]      # Temperature of the formation at RKB

    for j in range(1, well.zstep):

        if j <= well.riser:
            Tg = well.wtg    # Water Thermal Gradient for the Riser section
        else:
            Tg = well.gt      # Geothermal Gradient below the Riser section

        deltaT = Tsro[j - 1] + Tg*(well.tvd[j]-well.tvd[j-1])/well.deltaz

        # Generating the Temperature Profile at t=0
        Tfto.append(deltaT)
        Tto.append(deltaT)
        Tao.append(deltaT)
        Tco.append(deltaT)
        Tsro.append(deltaT)
        Tfm.append(deltaT)

    class InitCond(object):
        def __init__(self):
            self.tfto = Tfto
            self.tto = Tto
            self.tao = Tao
            self.tco = Tco
            self.tsro = Tsro
            self.tfm = Tfm

    return InitCond()
```

C.3. Heatcoefficients.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/production/heatcoefficients.py

```
def heat_coef(well, deltat, tt, t3):
    """
    Calculate heat transfer coefficients for each cell.
    :param t3: current temperature profile at section 3 (first casing)
    :param tt: current temperature profile at tubing wall
    :param well: a well object created from the function set_well()
    :param deltat: duration of each time step (seconds)
    :return: list with distribution of heat transfer coefficients
    """

    from math import pi, log
    from numpy import interp

    sections = [well.wd]
    if len(well.casings) > 0 and well.casings[0, 2] > 0:
        for i in range(len(well.casings))[::-1]:
            sections.append(well.casings[i, 2])

    vb = well.q / (pi * well.r3 ** 2)
    cbz = ((well.rhof[-1] * well.cf * vb) / well.deltaz) / 2  # Vertical component (North-South)
    cbe = (2 * well.h1[-1] / well.r3) / 2  # East component
    cbt = well.rhof[-1] * well.cf / deltat  # Time component

    # Creating empty lists

    # Section 1: Fluid in Tubing
    c1z = []
    c1e = []
    c1 = []
    c1t = []

    # Section 2: Tubing Wall
    c2z = []
    c2e = []
    c2w = []
    c2t = []

    # Section 3: Fluid in Annulus
    c3z = []
    c3e = []
    c3w = []
    c3t = []

    # Section 4: First casing
    c4z = []
```

```
c4e = []
c4w = []
c4t = []

# Section 5: Surrounding Space
c5z = []
c5e = []
c5w = []
c5t = []

in_section = 1
section_checkpoint = sections[0]

for x in range(well.zstep):
    if x*well.deltaz >= section_checkpoint and in_section < len(sections)+1:
        in_section += 1
        if section_checkpoint != sections[-1]:
            section_checkpoint = sections[in_section-1]

    gr_t = 9.81 * well.alpha * abs((tt[x] - t3[x])) * (well.rhof[x] ** 2) * (well.dti ** 3) / (well.visc ** 2)
    gr_c = 9.81 * well.alpha * abs((tt[x] - t3[x])) * (well.rhof[x] ** 2) * (((well.r3 - well.r2) * 2) ** 3) / (
            well.visc ** 2)
    ra_t = gr_t * well.pr
    ra_c = gr_c * well.pr
    inc = [0, 30, 45, 60, 90]
    c_base = [0.069, 0.065, 0.059, 0.057, 0.049]
    c = interp(well.inclination[x], inc, c_base, right=0.049)
    nu_a_t = c * (ra_t ** (1/3)) * (well.pr ** 0.074)
    nu_a_c = c * (ra_c ** (1/3)) * (well.pr ** 0.074)
    h2 = well.lambdaf * nu_a_t / (well.r2 * log(well.r3/well.r2))
    h3 = well.lambdaf * nu_a_c / (well.r2 * log(well.r3/well.r2))
    h3r = h3
    lambdal_eq = well.lambdaf * nu_a_t

    # fluid inside tubing
    qp = 0.2 * well.q * 2 * (well.f_p[x] * well.rhof[x] * (well.vp ** 2) *
                       (well.md[-1] / (well.dti * 127.094 * 10 ** 6)))

    c1z.append(((well.rhof[x] * well.cf * well.vp) / well.deltaz) / 2)  # Vertical component (North-South)
    c1e.append((2 * well.h1[x] / well.r1) / 2)  # East component
    c1.append(qp / (pi * (well.r1 ** 2)))  # Heat source term
    c1t.append(well.rhof[x] * well.cf / deltat)  # Time component

    # tubing wall
    c2z.append((well.lambdat / (well.deltaz ** 2)) / 2)  # Vertical component (North-South)
    c2e.append((2 * well.r2 * h2 / ((well.r2 ** 2) - (well.r1 ** 2))) / 2)  # East component
    c2w.append((2 * well.r1 * well.h1[x] / ((well.r2 ** 2) - (well.r1 ** 2))) / 2)  # West component
    c2t.append(well.rhot * well.ct / deltat)  # Time component

    if in_section == 1:
```

```
    lambda4 = well.lambdar  # Thermal conductivity of the casing (riser in this section)
    lambda5 = well.lambdaw  # Thermal conductivity of the surrounding space (seawater)
    lambda45 = (lambda4 * (well.r4r - well.r3r) + lambda5 * (well.r5 - well.r4r)) / (
            well.r5 - well.r3r)  # Comprehensive Thermal conductivity of the casing (riser) and
                             # surrounding space (seawater)
    lambda56 = well.lambdaw  # Comprehensive Thermal conductivity of the surrounding space (seawater) and
                           # formation (seawater)
    c4 = well.cr  # Specific Heat Capacity of the casing (riser)
    c5 = well.cw  # Specific Heat Capacity of the surrounding space (seawater)
    rho4 = well.rhor  # Density of the casing (riser)
    rho5 = well.rhow  # Density of the surrounding space (seawater)

    # fluid inside annular
    c3z.append((lambdal_eq / (well.deltaz ** 2)) / 2)  # Vertical component (North-South)
    c3e.append((2 * well.r3 * h3 / ((well.r3 ** 2) - (well.r2 ** 2))) / 2)  # East component
    c3w.append((2 * well.r2 * h2 / ((well.r3 ** 2) - (well.r2 ** 2))) / 2)  # West component
    c3t.append(well.rhof_a[x] * well.cf / deltat)  # Time component

else:
    # fluid inside annular
    c3z.append((lambdal_eq / (well.deltaz ** 2)) / 2)  # Vertical component (North-South)
    c3e.append((2 * well.r3 * h3r / ((well.r3r ** 2) - (well.r2 ** 2))) / 2)  # East component
    c3w.append((2 * well.r2 * h2 / ((well.r3r ** 2) - (well.r2 ** 2))) / 2)  # West component
    c3t.append(well.rhof_a[x] * well.cf / deltat)  # Time component

if 1 < in_section < len(sections):

    # calculation for surrounding space
    # thickness
    tcsr = 0
    tcem = 0
    for i in range(len(well.casings) - in_section):
        tcsr += (well.casings[i + 1, 0] - well.casings[i + 1, 1]) / 2
        tcem += (well.casings[i + 1, 1] - well.casings[i, 0]) / 2

        tcem += (well.casings[len(well.casings) - in_section + 1, 1] -
                 well.casings[len(well.casings) - in_section, 0]) / 2
    if in_section == 2:
        tcem += (well.dsro - well.casings[-1, 0])
    xcsr = tcsr / (well.r5 - well.r4)  # fraction of surrounding space that is casing
    xcem = tcem / (well.r5 - well.r4)  # fraction of surrounding space that is cement
    xfm = 1 - xcsr - xcem  # fraction of surrounding space that is formation

    # thermal conductivity
    lambdasr = well.lambdac * xcsr + well.lambdacem * xcem + well.lambdafm * xfm
    lambdacsr = (well.lambdac * (well.r4 - well.r3) + lambdasr * (well.r5 - well.r4)) / (well.r5 - well.r3)
    lambdasrfm = (well.lambdac * (well.r5 - well.r4) + lambdasr *   (well.rfm - well.r5)) / (well.rfm - well.r4)

    # Specific Heat Capacity
    csr = (well.cc * tcsr + well.ccem * tcem) / (well.r5 - well.r4)
```

```
        # Density
        rhosr = xcsr * well.rhoc + xcem * well.rhocem + xfm * well.rhofm

        lambda4 = well.lambdac
        lambda45 = lambdacsr
        lambda5 = lambdasr
        lambda56 = lambdasrfm
        c4 = well.cc  # Specific Heat Capacity of the casing
        c5 = csr  # Specific Heat Capacity of the surrounding space
        rho4 = well.rhoc  # Density of the casing
        rho5 = rhosr  # Density of the surrounding space

    if in_section == len(sections)+1:
        lambda4 = well.lambdafm
        lambda45 = well.lambdafm
        lambda5 = well.lambdafm
        lambda56 = well.lambdafm
        c4 = well.cfm  # Specific Heat Capacity of the casing (formation)
        c5 = well.cfm  # Specific Heat Capacity of the surrounding space (formation)
        rho4 = well.rhofm  # Density of the casing (formation)
        rho5 = well.rhofm  # Density of the surrounding space (formation)

    # first casing wall
    c4z.append((lambda4 / (well.deltaz ** 2)) / 2)
    c4e.append((2 * lambda45 / ((well.r4 ** 2) - (well.r3 ** 2))) / 2)
    c4w.append((2 * well.r3 * h3 / ((well.r4 ** 2) - (well.r3 ** 2))) / 2)
    c4t.append(rho4 * c4 / deltat)

    # surrounding space
    c5z.append((lambda5 / (well.deltaz ** 2)) / 2)
    c5w.append((lambda56 / (well.r5 * (well.r5 - well.r4) * log(well.r5 / well.r4))) / 2)
    c5e.append((lambda56 / (well.r5 * (well.r5 - well.r4) * log(well.rfm / well.r5))) / 2)
    c5t.append(rho5 * c5 / deltat)

hc_1 = [c1z, c1e, c1, c1t]
hc_2 = [c2z, c2e, c2w, c2t]
hc_3 = [c3z, c3e, c3w, c3t]
hc_4 = [c4z, c4e, c4w, c4t]
hc_5 = [c5z, c5e, c5w, c5t]
coefficients = [hc_1, hc_2, hc_3, hc_4, hc_5, cbe, cbt, cbz]

return coefficients
```

C.4. Linearsystem.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/production/linearsystem.py

```
def define_coef(coefficients, zstep):
    """
    Retrieves respective heat transfer coefficients for certain depth point.
    :param coefficients: list with distribution of heat transfer coefficients
    :param zstep: depth step
    :return: values of heat coefficients for each section at the same depth
    """

    hc_1 = coefficients[0]
    c1z = hc_1[0][zstep]
    c1e = hc_1[1][zstep]
    c1 = hc_1[2][zstep]
    c1t = hc_1[3][zstep]

    hc_2 = coefficients[1]
    c2z = hc_2[0][zstep]
    c2e = hc_2[1][zstep]
    c2w = hc_2[2][zstep]
    c2t = hc_2[3][zstep]

    hc_3 = coefficients[2]
    c3z = hc_3[0][zstep]
    c3e = hc_3[1][zstep]
    c3w = hc_3[2][zstep]
    c3t = hc_3[3][zstep]

    hc_4 = coefficients[3]
    c4z = hc_4[0][zstep]
    c4e = hc_4[1][zstep]
    c4w = hc_4[2][zstep]
    c4t = hc_4[3][zstep]

    hc_5 = coefficients[4]
    c5z = hc_5[0][zstep]
    c5e = hc_5[1][zstep]
    c5w = hc_5[2][zstep]
    c5t = hc_5[3][zstep]

    cbe = coefficients[5]
    cbt = coefficients[6]
    cbz = coefficients[7]

    return c1z, c1e, c1, c1t, c2z, c2e, c2w, c2t, c3z, c3e, c3w, c3t, c4z, c4e, c4w, c4t, c5z, c5e, c5w, c5t, cbe, \
        cbt, cbz
```

```
def temp_calc(well, initcond, heatcoeff):
    """
    Build the penta-diagonal matrix and solve it to get the well temperature distribution.
    :param well: a well object created from the function set_well()
    :param initcond: object with initial temperature profiles
    :param heatcoeff: list with distribution of heat transfer coefficients
    :return: object with final well temperature distribution
    """

    from numpy import zeros, linalg

    Tft = []
    Tt = []
    Ta = []
    tc = []
    Tsr = []
    xi = 5

    # Creating vectors N,W,C,E,S,B
    N = []
    W = []
    C = []
    E = []
    S = []
    B = []

    for j in range(well.zstep):
        c1z, c1e, c1, c1t, c2z, c2e, c2w, c2t, c3z, c3e, c3w, c3t, c4z, c4e, c4w, c4t, c5z, c5e, c5w, c5t, cbe, \
        cbt, cbz = define_coef(heatcoeff, j)
        for i in range(xi):

            if i == 0:  # Inside Tubing
                if j == 0:
                    C.append(c1t + c1e + c1z)
                    E.append(-c1e)
                    S.append(-c1z)
                    B.append(c1t * initcond.tfto[j]    # Center(t=0)
                             + c1   # Heat Source
                             + c1e * (initcond.tto[j] - initcond.tfto[j])     # East(t=0)
                             + c1z * (initcond.tfto[j + 1] - initcond.tfto[j]))     # N/S(t=0)

                if 0 < j < well.zstep - 2:
                    N.append(0)
                    W.append(0)
                    C.append(c1t + c1e + c1z)
                    E.append(-c1e)
                    S.append(-c1z)
                    B.append(c1t * initcond.tfto[j]    # Center(t=0)
                             + c1      # Heat Source
                + c1e * (initcond.tto[j] - initcond.tfto[j])      # East(t=0)
                + c1z * (initcond.tfto[j + 1] - initcond.tfto[j]))       # N/S(t=0)

        if j == well.zstep - 2:
            N.append(0)
            W.append(0)
            C.append(c1t + c1e + c1z)
            E.append(-c1e)
            S.append(0)
            B.append(c1t * initcond.tfto[j]  # Center(t=0)
                + c1  # Heat Source
                + c1e * (initcond.tto[j] - initcond.tfto[j])  # East(t=0)
                + c1z * (initcond.tfto[j + 1] - initcond.tfto[j])  # N/S(t=0)
                + c1z * initcond.tfto[-1])

        if j == well.zstep - 1:
            N.append(0)
            W.append(0)
            C.append(cbt + cbe)
            E.append(0)
            B.append(cbt * initcond.tfto[j]     # Center(t=0)
                + cbe * (initcond.tto[j] - initcond.tfto[j])      # East(t=0)
                + cbe * initcond.tto[-1])

    if i == 1:  # Tubing wall

        if j == 0:
            W.append(-c2w)
            C.append(c2t + c2e + c2w + c2z)
            E.append(-c2e)
            S.append(-c2z)
            B.append(c2t * initcond.tto[j]
                + c2e * (initcond.tao[j] - initcond.tto[j])
                + c2w * (initcond.tfto[j] - initcond.tto[j])
                + c2z * (initcond.tto[j + 1] - initcond.tto[j]))

        if 0 < j < well.zstep - 1:
            N.append(-c2z)
            W.append(-c2w)
            C.append(c2t + c2e + c2w + 2 * c2z)
            E.append(-c2e)
            if j < well.zstep - 3:
                S.append(-c2z)
                B.append(c2t * initcond.tto[j]
                    + c2e * (initcond.tao[j] - initcond.tto[j])
                    + c2w * (initcond.tfto[j] - initcond.tto[j])
                    + c2z * (initcond.tto[j + 1] - initcond.tto[j])
                    + c2z * (initcond.tto[j - 1] - initcond.tto[j]))
            else:
                S.append(0)
            B.append(c2t * initcond.tto[j]
                + c2e * (initcond.tao[j] - initcond.tto[j])
                + c2w * (initcond.tfto[j] - initcond.tto[j])
                + c2z * (initcond.tto[j + 1] - initcond.tto[j])
                + c2z * (initcond.tto[j - 1] - initcond.tto[j])
                + c2z * initcond.tfto[-1])

        if j == well.zstep - 1:
            N.append(-c2z)
            W.append(0)
            C.append(c2t + c2e + c2w + c2z)
            E.append(0)
            B.append(c2t * initcond.tto[j]
                    + c2e * (initcond.tao[j] - initcond.tto[j])
                    + c2w * initcond.tfto[j]
                    + c2e * initcond.tao[j]
                    + c2w * (initcond.tfto[j] - initcond.tto[j])
                    + c2z * (initcond.tto[j - 1] - initcond.tto[j]))

    if i == 2:  # Annular

        if j == 0:
            W.append(-c3w)
            C.append(c3t + c3e + c3w + c3z)
            E.append(-c3e)
            S.append(-c3z)
            B.append(c3t * initcond.tao[j]
                + c3e * (initcond.tco[j] - initcond.tao[j])
                + c3w * (initcond.tto[j] - initcond.tao[j])
                + c3z * (initcond.tao[j + 1] - initcond.tao[j]))

        if 0 < j < well.zstep - 1:
            N.append(-c3z)
            W.append(-c3w)
            C.append(c3t + c3e + c3w + 2 * c3z)
            E.append(-c3e)
            if j < well.zstep - 3:
                S.append(-c3z)
                B.append(c3t * initcond.tao[j]
                    + c3e * (initcond.tco[j] - initcond.tao[j])
                    + c3w * (initcond.tto[j] - initcond.tao[j])
                    + c3z * (initcond.tao[j + 1] - initcond.tao[j])
                    + c3z * (initcond.tao[j - 1] - initcond.tao[j]))
            else:
                S.append(0)
                B.append(c3t * initcond.tao[j]
                    + c3e * (initcond.tco[j] - initcond.tao[j])
                    + c3w * (initcond.tto[j] - initcond.tao[j])
                    + c3z * (initcond.tao[j + 1] - initcond.tao[j])
                    + c3z * (initcond.tao[j - 1] - initcond.tao[j])
                    + c3z * initcond.tfto[-1])

        if j == well.zstep - 1:
            N.append(-c3z)
            W.append(0)
            C.append(c3t + c3e + c3w + c3z)
            E.append(0)
            B.append(c3t * initcond.tao[j]
                    + c3e * (initcond.tao[j] - initcond.tto[j])
                    + c3w * (initcond.tfto[j] - initcond.tto[j])
                    + c3e * initcond.tco[j]
                    + c3w * initcond.tto[j]
                    + c3z * (initcond.tto[j - 1] - initcond.tto[j]))

    if i == 3:  # Casing

        if j == 0:
            W.append(-c4w)
            C.append(c4t + c4e + c4w + c4z)
            E.append(-c4e)
            S.append(-c4z)
            B.append(c4t * initcond.tco[j]   # Center(t=0)
                  + c4e * (initcond.tsro[j] - initcond.tco[j])  # East(t=0)
                  + c4w * (initcond.tao[j] - initcond.tco[j])  # West(t=0)
                  + c4z * (initcond.tco[j + 1] - initcond.tco[j]))   # N/S(t=0)

        if 0 < j < well.zstep - 2:
            N.append(-c4z)
            W.append(-c4w)
            C.append(c4t + c4e + c4w + 2 * c4z)
            E.append(-c4e)
            S.append(-c4z)
            B.append(c4t * initcond.tco[j]    # Center(t=0)
                  + c4e * (initcond.tsro[j] - initcond.tco[j])     # East(t=0)
                  + c4w * (initcond.tao[j] - initcond.tco[j])      # West(t=0)
                  + c4z * (initcond.tco[j + 1] - 2 * initcond.tco[j] + initcond.tco[j - 1])) # N/S(t=0)

        if j == well.zstep - 2:
            N.append(-c4z)
            W.append(-c4w)
            C.append(c4t + c4e + c4w + 2 * c4z)
            E.append(- c4e)
            S.append(0)
            B.append(c4t * initcond.tco[j]  # Center(t=0)
                  + c4e * (initcond.tsro[j] - initcond.tco[j])  # East(t=0)
                  + c4w * (initcond.tao[j] - initcond.tco[j])  # West(t=0)
                  + c4z * (initcond.tco[j + 1] - 2 * initcond.tco[j] + initcond.tco[j - 1])  # N/S(t=0)
                  + c4z * initcond.tfto[-1])

        if j == well.zstep - 1:
```

```
        N.append(-c4z)
        W.append(0)
        C.append(c4t + c4e + c4w + c4z)
        E.append(0)
        B.append(c4t * initcond.tco[j]   # Center(t=0)
                 + c4e * (initcond.tsro[j] - initcond.tco[j])    # East(t=0)
                 + c4w * (initcond.tao[j] - initcond.tco[j])     # West(t=0)
                 + c4z * (initcond.tco[j - 1] - initcond.tco[j])     # N/S(t=0)
                 + c4z * initcond.tfto[-1]
                 + c4w * initcond.tfto[-1])

if i == 4:  # Surrounding Space

    if j == 0:
        W.append(-c5w)
        C.append(c5w + c5z + c5e + c5t)
        E.append(-c5e)
        S.append(-c5z)
        B.append(c5w * (initcond.tco[j] - initcond.tsro[j])
                 + c5z * (initcond.tsro[j + 1] - initcond.tsro[j])
                 + c5t * initcond.tsro[j])

    if 0 < j < well.zstep - 2:
        N.append(-c5z)
        W.append(-c5w)
        C.append(c5w + c5e + 2 * c5z + c5t)
        E.append(-c5e)
        S.append(-c5z)
        B.append(c5w * (initcond.tco[j] - initcond.tsro[j])
                 + c5z * (initcond.tsro[j + 1] - initcond.tsro[j])
                 + c5z * (initcond.tsro[j - 1] - initcond.tsro[j])
                 + c5t * initcond.tsro[j])

    if j == well.zstep - 2:
        N.append(-c5z)
        W.append(-c5w)
        C.append(c5w + c5e + 2 * c5z + c5t)
        E.append(-c5e)
        S.append(0)
        B.append(c5w * (initcond.tco[j] - initcond.tsro[j])
                 + c5z * (initcond.tsro[j + 1] - initcond.tsro[j])
                 + c5z * (initcond.tsro[j - 1] - initcond.tsro[j])
                 + c5t * initcond.tsro[j]
                 + c5z * initcond.tfto[-1])

    if j == well.zstep - 1:
        N.append(-c5z)
        W.append(0)
        C.append(c5w + c5e + c5z + c5t)
        B.append(c5w * (initcond.tco[j] - initcond.tsro[j])
                + c5z * (initcond.tsro[j - 1] - initcond.tsro[j])
                + c5z * initcond.tsro[j]
                + c5t * initcond.tsro[j])

#LINEARSYSTEM
# Creating pentadiagonal matrix
A = zeros((xi * well.zstep - 0, xi * well.zstep - 0))

# Filling up Pentadiagonal Matrix A
lenC = xi * well.zstep - 0
lenN = lenC - xi
lenW = lenC - 1
lenE = lenC - 1
lenS = lenC - xi

for it in range(lenC):  # Inserting list C
  A[it, it] = C[it]
for it in range(lenE):  # Inserting list E
  A[it, it + 1] = E[it]
for it in range(lenW):  # Inserting list W
  A[it + 1, it] = W[it]
for it in range(lenN):  # Inserting list N
  A[it + xi, it] = N[it]
for it in range(lenS):  # Inserting list S
  A[it, it + xi] = S[it]

Temp = linalg.solve(A, B)

for x in range(well.zstep):
  if x < well.zstep - 1:
    Tft.append(Temp[5 * x])
  if x == well.zstep - 1:
    Tft.append(initcond.tfto[-1])
for x in range(well.zstep):
  if x < well.zstep - 1:
    Tt.append(Temp[5 * x + 1])
  if x == well.zstep - 1:
    Tt.append(initcond.tto[-1])
for x in range(well.zstep):
  if x < well.zstep - 1:
    Ta.append(Temp[5 * x + 2])
  if x == well.zstep - 1:
    Ta.append(initcond.tao[-1])
for x in range(well.zstep):
  if x < well.zstep - 1:
    tc.append(Temp[5 * x + 3])
  if x == well.zstep - 1:
    tc.append(initcond.tco[-1])
for x in range(well.zstep):
  if x < well.zstep - 1:
        Tsr.append(Temp[5 * x + 4])
    if x == well.zstep - 1:
        Tsr.append(initcond.tsro[-1])

t3 = tc.copy()

tr = tc[:well.riser] + [None] * (well.zstep - well.riser)
for x in range(well.riser):
    tc[x] = None

csgs_reach = int(well.casings[0, 2] / well.deltaz)  # final depth still covered with casing(s)

Toh = [None] * csgs_reach + tc[csgs_reach:]
for x in range(csgs_reach, well.zstep):
    tc[x] = None

class TempCalc(object):
    def __init__(self):
        self.tft = Tft
        self.tt = Tt
        self.ta = Ta
        self.t3 = t3
        self.tc = tc
        self.tr = tr
        self.tsr = Tsr
        self.toh = Toh
        self.csgs_reach = csgs_reach

return TempCalc()
```

C.5. Fluid.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/production/fluid.py

```
def initial_density(well, initcond, section='tubing'):
    """
    Function to calculate the density profile for the first time step
    :param section: 'tubing' or 'annular'
    :param well: a well object created from the function set_well()
    :param initcond: a initial conditions object with the formation temperature profile
    :return: the density profile and the initial density at surface conditions
    """

    if section == 'tubing':
        beta = well.beta
        alpha = well.alpha
        rho = well.rhof

    if section == 'annular':
        beta = well.beta_a
        alpha = well.alpha_a
        rho = well.rhof_a

    rhof_initial = rho
    pressure = [rho * 9.81 * i for i in well.tvd]
    rhof = [rhof_initial * (1 + (x - 10 ** 5) / beta - alpha * (y - well.ts)) for x, y in
            zip(pressure, initcond.tfto)]
    pressure = [x * 9.81 * y for x, y in zip(rhof, well.tvd)]

    rhof = [rhof_initial * (1 + (x - 10 ** 5) / beta - alpha * (y - well.ts)) for x, y in
            zip(pressure, initcond.tfto)]

    return rhof, rhof_initial


def calc_density(well, initcond, rhof_initial, section='tubing'):
    """
    Function to calculate the density profile
    :param section: 'tubing' or 'annular'
    :param well: a well object created from the function set_well()
    :param initcond: a initial conditions object with the formation temperature profile
    :param rhof_initial: initial density at surface conditions
    :param flow: boolean to define if the section is flowing
    :return: density profile
    """

    if section == 'tubing':
        beta = well.beta
        alpha = well.alpha
    flow = True
    temp = initcond.tfto
    rho = well.rhof
if section == 'annular':
    beta = well.beta_a
    alpha = well.alpha_a
    flow = False
    temp = initcond.tao
    rho = well.rhof_a

pressure_h = [x * 9.81 * y for x, y in zip(rho, well.tvd)]

if flow:
    pressure_f = [x * (well.md[-1] / well.dti) * (1/2) * y * well.vp **2 for x, y in zip(well.f_p, rho)]
else:
    pressure_f = [0] * len(well.md)

pressure = [x + y for x, y in zip(pressure_h, pressure_f)]

rhof = [rhof_initial * (1 + (x - 10 ** 5) / beta - alpha * (y - well.ts)) for x, y in
        zip(pressure, temp)]

return rhof
```

C.6. Main.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/production/main.py

```
import numpy as np


def temp_time(n, well, log=True, units='metric', time_delta=None):
    """
    Function to calculate the well temperature distribution during certain production time (n)
    :param n: production time, hours
    :param well: a well object created with the function set_well() from input.py
    :param log: save distributions between initial time and circulation time n (each 1 hour)
    :param units: system of units ('metric' or 'english')
    :param time_delta: duration of each time step (seconds)
    :return: a well temperature distribution object
    """
    from .initcond import init_cond
    from .heatcoefficients import heat_coef
    from .linearsystem import temp_calc
    from .plot import profile
    from math import log, nan
    # Simulation main parameters
    time = n  # circulating time, h
    tcirc = time * 3600  # circulating time, s
    deltat = 60 * time
    if type(time_delta) == int:
        deltat = time_delta
    tstep = int(tcirc / deltat)
    ic = init_cond(well)
    tfm = ic.tfm
    tt = ic.tto
    t3 = ic.tco

    well = well.define_density(ic, cond=0)

    hc = heat_coef(well, deltat, tt, t3)
    temp = temp_calc(well, ic, hc)
    temp.tft = temp.tt = temp.ta = temp.t3 = temp.tsr = tfm
    for x in range(len(tfm)):
        if temp.tc[x] != nan:
            temp.tc[x] = tfm[x]
        if temp.tr[x] != nan:
            temp.tr[x] = tfm[x]
        if temp.toh[x] != nan:
            temp.toh[x] = tfm[x]

    temp_initial = temp
    temp_initial.tft = ic.tfm
temp_initial.tt = ic.tfm
temp_initial.ta = ic.tfm

temp_log = [temp_initial, temp]
time_log = [0, deltat / 3600]

for x in range(tstep-1):
    well = well.define_density(ic, cond=1)

    ic.tfto = temp.tft
    ic.tto = temp.tt
    ic.tao = temp.ta
    ic.tco = temp.t3
    ic.tsr = temp.tsr
    hc_new = heat_coef(well, deltat, ic.tto, ic.tco)
    temp = temp_calc(well, ic, hc_new)

    if units == 'english':
        temp.tft_output = [(i/(5/9)+32) for i in temp.tft]
        temp.tt_output = [(i/(5/9)+32) for i in temp.tt]
        temp.ta_output = [(i/(5/9)+32) for i in temp.ta]
        temp.tc_output = [(i/(5/9)+32) for i in temp.tc if type(i) == np.float64]
        temp.tr_output = [(i/(5/9)+32) for i in temp.tr if type(i) == np.float64]
        temp.tsr_output = [(i/(5/9)+32) for i in temp.tsr]
        temp.md_output = [i*3.28 for i in well.md]

    if log:
        temp_log.append(temp)
        time_log.append(time_log[-1] + time_log[1])

if units == 'english':
    temp.tft = temp.tft_output
    temp.tt = temp.tt_output
    temp.ta = temp.ta_output
    temp.tc = temp.tc_output
    temp.tr = temp.tr_output
    temp.sr = temp.tsr_output
    temp.md = temp.md_output
    tfm = [(i / (5 / 9) + 32) for i in tfm]

class TempDist(object):
    def __init__(self):
        self.tft = temp.tft
        self.tt = temp.tt
        self.ta = temp.ta
        self.tc = temp.tc
        self.tr = temp.tr
        self.tsr = temp.tsr
        self.tfm = tfm
        self.time = time
```

```
            self.md = well.md
            self.riser = well.riser
            self.deltat = deltat
            self.csgs_reach = temp.csgs_reach
            if log:
                self.temp_log = temp_log
                self.time_log = time_log

        def well(self):
            return well

        def plot(self, tft=True, tt=False, ta=True, tc=False, tr=False, sr=False):
            profile(self, tft, tt, ta, tc, tr, sr, units)

        def behavior(self):
            temp_behavior_production = temp_behavior(self)
            return temp_behavior_production

        def plot_multi(self, tft=True, ta=False, tr=False, tc=False, tfm=False, tsr=False):
            plot_multitime(self, tft, ta, tr, tc, tfm, tsr)

    return TempDist()


def temp_behavior(temp_dist):

    ta = [x.ta for x in temp_dist.temp_log]

    tout = []

    for n in range(len(ta)):
        tout.append(ta[n][0])

    class Behavior(object):
        def __init__(self):
            self.finaltime = temp_dist.time
            self.tout = tout
            self.tfm = temp_dist.tfm
            self.time = temp_dist.time_log

        def plot(self):
            from .plot import behavior
            behavior(self)

    return Behavior()


def plot_multitime(temp_dist, tft=True, ta=False, tr=False, tc=False, tfm=False, tsr=False):
    from .plot import profile_multitime
    values = temp_dist.temp_log
    times = [x for x in temp_dist.time_log]
    profile_multitime(temp_dist, values, times, tft=tft, ta=ta, tr=tr, tc=tc, tfm=tfm, tsr=tsr)


def temp(n, mdt=3000, casings=[], wellpath_data=[], d_openhole=0.216, grid_length=50, profile='V', build_angle=1,
kop=0,
         eob=0, sod=0, eod=0, kop2=0, eob2=0, change_input={}, log=False, units='metric', time_delta=None):
    """
    Main function to calculate the well temperature distribution during production operation. This function allows to
    set the wellpath and different parameters involved.
    :param n: production time, hours
    :param mdt: measured depth of target, m
    :param casings: list of dictionaries with casings characteristics (od, id and depth)
    :param wellpath_data: load own wellpath as a list
    :param d_openhole: diameter of open hole section, m
    :param grid_length: number of cells through depth
    :param profile: type of well to generate. Vertical ('V'), S-type ('S'), J-type ('J') and Horizontal ('H1' or 'H2')
    :param build_angle: build angle, °
    :param kop: kick-off point, m
    :param eob: end of build, m
    :param sod: start of drop, m
    :param eod: end of drop, m
    :param kop2: kick-off point 2, m
    :param eob2: end of build 2, m
    :param change_input: dictionary with parameters to set.
    :param log: save distributions between initial time and circulation time n (each 1 hour)
    :param units: system of units ('metric' or 'english')
    :param time_delta: duration of each time step (seconds)
    :return: a well temperature distribution object
    """
    from .input import data, set_well
    from .. import wellpath
    tdata = data(casings, d_openhole, units)
    for x in change_input:   # changing default values
        if x in tdata:
            tdata[x] = change_input[x]
        else:
            raise TypeError('%s is not a parameter' % x)
    if len(wellpath_data) == 0:
        depths = wellpath.get(mdt, grid_length, profile, build_angle, kop, eob, sod, eod, kop2, eob2, units)
    else:
        depths = wellpath.load(wellpath_data, grid_length, units)
    well = set_well(tdata, depths, units)
    temp_distribution = temp_time(n, well, log, units, time_delta)

    return temp_distribution


def input_info(about='all'):
```

```
from .input import info
info(about)
```

Appendix D Injection Module

D.1. Input.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/injection/input.py

```
def data(casings=[], d_openhole=0.216, units='metric'):
    """
    Parameters involved within the operation calculations
    :param casings: list of dictionaries with casings characteristics (od, id and depth)
    :param d_openhole: diameter of open hole section, m
    :param units: system of units ('metric' or 'english')
    :return: a dictionary with default values for the required parameters
    """

    from numpy import asarray

    dict_met = {'ts': 15.0, 'wd': 100.0,  'dti': 4.0, 'dto': 4.5, 'dri': 17.716, 'dro': 21.0, 'dfm': 80.0,
            'q': 144, 'lambdaf': 0.635, 'lambdac': 43.3, 'lambdacem': 0.7, 'lambdat': 40.0, 'lambdafm': 2.249,
            'lambdar': 15.49, 'lambdaw': 0.6, 'cf': 3713.0, 'cc': 469.0, 'ccem': 2000.0, 'ct': 400.0, 'cr': 464.0,
            'cw': 4000.0, 'cfm': 800.0, 'rhof': 1.198, 'rhof_a': 1.2, 'rhot': 7.6, 'rhoc': 7.8, 'rhor': 7.8,
            'rhofm': 2.245, 'rhow': 1.029, 'rhocem': 2.7, 'gt': 0.0238, 'wtg': -0.005, 'visc': 1, 'tin': 20,
            'beta': 44983 * 10 ** 5, 'alpha': 960 * 10 ** -6, 'beta_a': 44983 * 10 ** 5, 'alpha_a': 960 * 10 ** -6}

    dict_eng = {'ts': 59.0, 'wd': 328.0, 'dti': 4.0, 'dto': 4.5, 'dri': 17.716, 'dro': 21.0, 'dfm': 80.0,
            'q': 26.42, 'lambdaf': 1.098, 'lambdac': 74.909, 'lambdacem': 1.21, 'lambdat': 69.2, 'lambdafm': 3.89,
            'lambdar': 26.8, 'lambdaw': 1.038, 'cf': 0.887, 'cc': 0.112, 'ccem': 0.478, 'ct': 0.096, 'cr': 0.1108,
            'cw': 0.955, 'cfm': 0.19, 'rhof': 9.997, 'rhof_a': 10, 'rhot': 65.09, 'rhoc': 65.09, 'rhor': 65.09,
            'rhofm': 18.73, 'rhow': 8.587, 'rhocem': 22.5, 'gt': 0.00403, 'wtg': -8.47*10**-4, 'visc': 1, 'tin': 68,
            'beta': 652423, 'alpha': 5.33 * 10 ** -4, 'beta_a': 652423, 'alpha_a': 5.33 * 10 ** -4}

    if units == 'metric':
        dict = dict_met
    else:
        dict = dict_eng

    if len(casings) > 0:
        od = sorted([x['od'] * 0.0254 for x in casings])
        id = sorted([x['id'] * 0.0254 for x in casings])
        depth = sorted([x['depth'] for x in casings], reverse=True)
        dict['casings'] = [[od[x], id[x], depth[x]] for x in range(len(casings))]
        dict['casings'] = asarray(dict['casings'])
    else:
        dict['casings'] = [[(d_openhole + dict['dro'] * 0.0254), d_openhole, 0]]
        dict['casings'] = asarray(dict['casings'])

    return dict
```

```
def info(about='all'):
    """
    Retrieves information about the parameters (description and units)
    :param about: type of parameters
    :return: description and units of parameters
    """

    print("Use the ID of a parameter to change the default value (e.g. tdict['tin']=30 to change the fluid inlet "
          "temperature from the default value to 30° Celsius)")
    print('Notice that the information is provided as follows:' + '\n' +
          'parameter ID: general description, units' + '\n')

    tubular_parameters = 'VALUES RELATED TO TUBULAR SIZES' + '\n' + \
                         'dti: tubing inner diameter, in' + '\n' + \
                         'dto: tubing outer diameter, in' + '\n' + \
                         'dri: riser inner diameter, in' + '\n' + \
                         'dro: riser outer diameter, in' + '\n'

    conditions_parameters = 'PARAMETERS RELATED TO SIMULATION CONDITIONS' + '\n' + \
                            'ts: surface temperature, °C or °F' + '\n' + \
                            'wd: water depth, m or ft' + '\n' + \
                            'dfm: undisturbed formation diameter, m or ft' + '\n'

    heatcoeff_parameters = 'PARAMETERS RELATED TO HEAT COEFFICIENTS' + '\n' + \
                           'lambdaf: fluid - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdac: casing - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdacem: cement - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdat: tubing - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdafm: formation - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdar: riser - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'lambdaw: water - thermal conductivity, W/(m*°C) or BTU/(h*ft*°F)' + '\n' + \
                           'cf: fluid - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'cc: casing - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'ccem: cement - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'ct: tubing - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'cr: riser - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'cw: water - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'cfm: formation - specific heat capacity, J/(kg*°C) or BTU/(lb*°F)' + '\n' + \
                           'gt: geothermal gradient, °C/m or °F/ft' + '\n' + \
                           'wtg: seawater thermal gradient, °C/m or °F/ft' + '\n'

    densities_parameters = 'PARAMETERS RELATED TO DENSITIES' + '\n' + \
                           'rhof: fluid density, sg or ppg' + '\n' + \
                           'rhot: tubing density, sg or ppg' + '\n' + \
                           'rhoc: casing density, sg or ppg' + '\n' + \
                           'rhor: riser density, sg or ppg' + '\n' + \
                           'rhofm: formation density, sg or ppg' + '\n' + \
                           'rhow: seawater density, sg or ppg' + '\n' + \
                           'rhocem: cement density, sg or ppg' + '\n' + \
                           'beta: isothermal bulk modulus of injection fluid, Pa' + '\n' + \
                           'alpha: expansion coefficient of injection fluid, 1/°C' + '\n' + \
                           'beta_a: isothermal bulk modulus of fluid in annular, Pa or psi' + '\n' + \
                           'alpha_a: expansion coefficient of fluid in annular, 1/°C or 1/°F' + '\n'

    viscosity_parameters = 'PARAMETERS RELATED TO MUD VISCOSITY' + '\n' + \
                           'thao_o: yield stress, Pa or psi' + '\n' + \
                           'n: flow behavior index, dimensionless' + '\n' + \
                           'k: consistency index, Pa*s^n or psi*s^n' + '\n' + \
                           'visc: fluid viscosity, cp' + '\n'

    operational_parameters = 'PARAMETERS RELATED TO THE OPERATION' + '\n' + \
                     'tin: fluid inlet temperature, °C or °F' + '\n' + \
                     'q: flow rate, m^3/day or gpm' + '\n'

    if about == 'casings':
        print(tubular_parameters)

    if about == 'conditions':
        print(conditions_parameters)

    if about == 'heatcoeff':
        print(heatcoeff_parameters)

    if about == 'densities':
        print(densities_parameters)

    if about == 'operational':
        print(operational_parameters)

    if about == 'viscosity':
        print(viscosity_parameters)

    if about == 'all':
        print(tubular_parameters + '\n' + conditions_parameters + '\n' + heatcoeff_parameters + '\n' +
              densities_parameters + '\n' + viscosity_parameters + '\n' + operational_parameters)


def set_well(temp_dict, depths, units='metric'):
    """
    Define properly the parameters and respective values within an object well.
    :param temp_dict: dictionary with inputs and default values.
    :param depths: wellpath object
    :param units: system of units ('metric' or 'english')
    :return: a well object with conditions and parameters defined
    """

    from math import pi, log
```

```
def wellpath():
    """
    :return: wellpath object
    """
    return depths

class NewWell(object):
    def __init__(self):
        # DEPTH
        self.md = depths.md
        self.tvd = depths.tvd
        self.deltaz = depths.deltaz
        self.zstep = depths.zstep
        self.sections = depths.sections
        self.north = depths.north
        self.east = depths.east
        self.inclination = depths.inclination
        self.dogleg = depths.dogleg
        self.azimuth = depths.azimuth
        if units != 'metric':
            self.md = [i / 3.28 for i in self.md]
            self.tvd = [i / 3.28 for i in self.tvd]
            self.deltaz = self.deltaz / 3.28
            self.north = [i / 3.28 for i in self.north]
            self.east = [i / 3.28 for i in self.east]

        # TUBULAR
        if units == 'metric':
            d_conv = 0.0254  # from in to m
        else:
            d_conv = 0.0254  # from in to m
        self.casings = temp_dict["casings"]  # casings array
        self.dti = temp_dict["dti"] * d_conv  # Tubing Inner  Diameter, m
        self.dto = temp_dict["dto"] * d_conv   # Tubing Outer Diameter, m
        self.dri = temp_dict["dri"] * d_conv  # Riser diameter Inner Diameter, m
        self.dro = temp_dict["dro"] * d_conv   # Riser diameter Outer Diameter, m

        # CONDITIONS
        if units == 'metric':
            depth_conv = 1  # from m to m
            self.ts = temp_dict["ts"]  # Surface Temperature (RKB), °C
        else:
            depth_conv = 1 / 3.28  # from ft to m
            self.ts = (temp_dict["ts"] - 32) * (5 / 9)  # Surface Temperature (RKB), from °F to °C
        self.wd = temp_dict["wd"] * depth_conv  # Water Depth, m
        self.riser = round(self.wd / self.deltaz)  # number of grid cells for the riser
        self.dsr = self.casings[0, 0]  # Surrounding Space Inner Diameter, m
        self.dsro = sorted([self.dro + 0.03, self.casings[-1, 0] + 0.03])[-1]  # Surrounding Space Outer Diameter, m
        self.dfm = temp_dict["dfm"]  # Undisturbed Formation Diameter, m
```

```
# RADIUS (CALCULATED)
self.r1 = self.dti / 2  # Tubing Inner  Radius, m
self.r2 = self.dto / 2  # Tubing Outer Radius, m
self.r3 = self.casings[0, 1] / 2  # Casing Inner Radius, m
self.r3r = self.dri / 2  # Riser Inner Radius, m
self.r4r = self.dro / 2  # Riser Outer Radius, m
self.r4 = self.casings[0, 0] / 2  # Surrounding Space Inner Radius m
self.r5 = self.dsro / 2  # Surrounding Space Outer Radius, m
self.rfm = self.dfm / 2  # Undisturbed Formation Radius, m

# DENSITIES kg/m3
if units == 'metric':
  dens_conv = 1000  # from sg to kg/m3
else:
  dens_conv = 119.83  # from ppg to kg/m3
self.rhof = temp_dict["rhof"] * dens_conv  # Fluid
self.rhof_a = temp_dict["rhof_a"] * dens_conv  # Fluid
self.rhot = temp_dict["rhot"] * dens_conv  # Tubing
self.rhoc = temp_dict["rhoc"] * dens_conv  # Casing
self.rhor = temp_dict["rhor"] * dens_conv  # Riser
self.rhocem = temp_dict["rhocem"] * dens_conv  # Cement Sheath
self.rhofm = temp_dict["rhofm"] * dens_conv  # Formation
self.rhow = temp_dict["rhow"] * dens_conv  # Seawater
self.visc = temp_dict["visc"] / 1000  # Fluid viscosity [Pas]

# OPERATIONAL
if units == 'metric':
  self.tin = temp_dict["tin"]  # Inlet Fluid temperature, °C
  q_conv = 0.04167     # from m^3/day to m^3/h
else:
  self.tin = (temp_dict["tin"] - 32) * (5/9)  # Inlet Fluid temperature, from °F to °C
  q_conv = 0.2271   # from gpm to m^3/h
self.q = temp_dict["q"] * q_conv  # Flow rate, m^3/h
self.vp = (self.q / (pi * (self.r1 ** 2))) / 3600  # Fluid velocity through the tubing

# HEAT COEFFICIENTS
if units == 'metric':
  lambda_conv = 1    # from W/(m*°C) to W/(m*°C)
  c_conv = 1  # from J/(kg*°C) to J/(kg*°C)
  gt_conv = 1    # from °C/m to °C/m
  beta_conv = 1   # from Pa to Pa
  alpha_conv = 1  # from 1/°F to 1/°C
else:
  lambda_conv = 1/1.73    # from BTU/(h*ft*°F) to W/(m*°C)
  c_conv = 4187.53  # from BTU/(lb*°F) to J/(kg*°C)
  gt_conv = 3.28*1.8    # from °F/ft to °C/m
  beta_conv = 6894.76  # from psi to Pa
  alpha_conv = 1.8  # from 1/°F to 1/°C

# Thermal conductivity  W/(m*°C)
```

```
        self.lambdaf = temp_dict["lambdaf"] * lambda_conv  # Fluid
        self.lambdac = temp_dict["lambdac"] * lambda_conv  # Casing
        self.lambdacem = temp_dict["lambdacem"] * lambda_conv  # Cement
        self.lambdat = temp_dict["lambdat"] * lambda_conv  # Tubing wall
        self.lambdafm = temp_dict["lambdafm"] * lambda_conv  # Formation
        self.lambdar = temp_dict["lambdar"] * lambda_conv  # Riser
        self.lambdaw = temp_dict["lambdaw"] * lambda_conv  # Seawater

        self.beta = temp_dict["beta"] * beta_conv  # isothermal bulk modulus in tubing, Pa
        self.alpha = temp_dict['alpha'] * alpha_conv  # Fluid Thermal Expansion Coefficient in tubing, 1/°C
        self.beta_a = temp_dict["beta_a"] * beta_conv  # isothermal bulk modulus in annular, Pa
        self.alpha_a = temp_dict['alpha_a'] * alpha_conv  # Fluid Thermal Expansion Coefficient in annular, 1/°C

        # Specific heat capacity, J/(kg*°C)
        self.cf = temp_dict["cf"] * c_conv  # Fluid
        self.cc = temp_dict["cc"] * c_conv  # Casing
        self.ccem = temp_dict["ccem"] * c_conv  # Cement
        self.ct = temp_dict["ct"] * c_conv  # Tubing
        self.cr = temp_dict["cr"] * c_conv  # Riser
        self.cw = temp_dict["cw"] * c_conv  # Seawater
        self.cfm = temp_dict["cfm"] * c_conv  # Formation

        self.pr = self.visc * self.cf / self.lambdaf  # Prandtl number

        self.gt = temp_dict["gt"] * gt_conv * self.deltaz  # Geothermal gradient, °C/m
        self.wtg = temp_dict["wtg"] * gt_conv * self.deltaz  # Seawater thermal gradient, °C/m

        # Raise Errors:

        if self.casings[-1, 0] > self.dsro:
            raise ValueError('Last casing outer diameter must be smaller than the surrounding space diameter.')

        if self.casings[0, 2] > self.md[-1]:
            raise ValueError('MD must be higher than the first casing depth.')

        if self.casings[0, 1] < self.dto:
            raise ValueError('Tubing outer diameter must be smaller than the first casing inner diameter.')

        if self.wd > 0 and self.dro > self.dsro:
            raise ValueError('Riser diameter must be smaller than the surrounding space diameter.')

        if self.dsro > self.dfm:
            raise ValueError('Surrounding space diameter must be smaller than the undisturbed formation diameter.')

    def define_density(self, ic, cond=0):
        """
        Calculate the density profile
        :param ic: current temperature distribution
        :param cond: '0' to calculate the initial profile
        :return: density profile and derived calculations
        """

        from .fluid import initial_density, calc_density

        if cond == 0:
            self.rhof, self.rhof_initial = initial_density(self, ic)
            self.rhof_a, self.rhof_a_initial = initial_density(self, ic, section='annular')
        else:
            self.rhof = calc_density(self, ic, self.rhof_initial)
            self.rhof_a = calc_density(self, ic, self.rhof_initial, section='annular')
        self.re_p = [x * self.vp * 2 * self.r1 / self.visc for x in self.rhof]  # Reynolds number inside tubing
        self.f_p = []  # Friction factor inside tubing
        self.nu_dpi = []
        for x in range(len(self.md)):
            if self.re_p[x] < 2300:
                self.f_p.append(64 / self.re_p[x])
                self.nu_dpi.append(4.36)
            else:
                self.f_p.append(1.63 / log(6.9 / self.re_p[x]) ** 2)
                self.nu_dpi.append(
                    (self.f_p[x] / 8) * (self.re_p[x] - 1000) * self.pr / (1 + (12.7 * (self.f_p[x] / 8) ** 0.5) *
                                                                           (self.pr ** (2 / 3) - 1)))
        # convective heat transfer coefficients, W/(m^2*°C)
        self.h1 = [self.lambdaf * x / self.dti for x in self.nu_dpi]  # Tubing inner wall
        return self

    return NewWell()
```

D.2. Initcond.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/injection/initcond.py

```
def init_cond(well):
    """
    Generates the temperature profiles at time 0, before starting the operation.
    :param well: a well object created from the function set_well()
    :return: object with initial temperature profiles
    """

    # Initial Conditions
    Tfto = [well.ts]   # Temperature of the fluid inside the tubing at RKB
    Tto = [well.ts]    # Temperature of the tubing at RKB, t=0
    Tao = [well.ts]      # Temperature of the fluid inside the annulus at RKB, t=0
    Tco = [well.ts]    # Temperature of the casing at RKB, t=0
    Tsro = [well.ts]    # Temperature of the surrounding space at RKB, t=0
    Tfm = [well.ts]      # Temperature of the formation at RKB

    for j in range(1, well.zstep):

        if j <= well.riser:
            Tg = well.wtg    # Water Thermal Gradient for the Riser section
        else:
            Tg = well.gt      # Geothermal Gradient below the Riser section

        deltaT = Tsro[j - 1] + Tg*(well.tvd[j]-well.tvd[j-1])/well.deltaz

        # Generating the Temperature Profile at t=0
        Tfto.append(deltaT)
        Tto.append(deltaT)
        Tao.append(deltaT)
        Tco.append(deltaT)
        Tsro.append(deltaT)
        Tfm.append(deltaT)

    class InitCond(object):
        def __init__(self):
            self.tfto = Tfto
            self.tto = Tto
            self.tao = Tao
            self.tco = Tco
            self.tsro = Tsro
            self.tfm = Tfm

    return InitCond()
```

D.3. Heatcoefficients.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/injection/heatcoefficients.py

```python
def heat_coef(well, deltat, tt, t3):
    """
    Calculate heat transfer coefficients for each cell.
    :param t3: current temperature profile at section 3 (first casing)
    :param tt: current temperature profile at tubing wall
    :param well: a well object created from the function set_well()
    :param deltat: duration of each time step (seconds)
    :return: list with distribution of heat transfer coefficients
    """

    from math import pi, log
    from numpy import interp

    sections = [well.wd]
    if len(well.casings) > 0 and well.casings[0, 2] > 0:
        for i in range(len(well.casings))[::-1]:
            sections.append(well.casings[i, 2])

    vb = well.q / (pi * well.r3 ** 2)
    cbz = ((well.rhof[-1] * well.cf * vb) / well.deltaz) / 2  # Vertical component (North-South)
    cbe = (2 * well.h1[-1] / well.r3) / 2  # East component
    cbt = well.rhof[-1] * well.cf / deltat  # Time component

    # Creating empty lists

    # Section 1: Fluid in Tubing
    c1z = []
    c1e = []
    c1 = []
    c1t = []

    # Section 2: Tubing Wall
    c2z = []
    c2e = []
    c2w = []
    c2t = []

    # Section 3: Fluid in Annulus
    c3z = []
    c3e = []
    c3w = []
    c3t = []

    # Section 4: First casing
    c4z = []
```

```
c4e = []
c4w = []
c4t = []

# Section 5: Surrounding Space
c5z = []
c5e = []
c5w = []
c5t = []

in_section = 1
section_checkpoint = sections[0]

for x in range(well.zstep):
    if x*well.deltaz >= section_checkpoint and in_section < len(sections)+1:
        in_section += 1
        if section_checkpoint != sections[-1]:
            section_checkpoint = sections[in_section-1]

    gr_t = 9.81 * well.alpha * abs((tt[x] - t3[x])) * (well.rhof[x] ** 2) * (well.dti ** 3) / (well.visc ** 2)
    gr_c = 9.81 * well.alpha * abs((tt[x] - t3[x])) * (well.rhof[x] ** 2) * (((well.r3 - well.r2) * 2) ** 3) / (
            well.visc ** 2)
    ra_t = gr_t * well.pr
    ra_c = gr_c * well.pr
    inc = [0, 30, 45, 60, 90]
    c_base = [0.069, 0.065, 0.059, 0.057, 0.049]
    c = interp(well.inclination[x], inc, c_base, right=0.049)
    nu_a_t = c * (ra_t ** (1/3)) * (well.pr ** 0.074)
    nu_a_c = c * (ra_c ** (1/3)) * (well.pr ** 0.074)
    h2 = well.lambdaf * nu_a_t / (well.r2 * log(well.r3/well.r2))
    h3 = well.lambdaf * nu_a_c / (well.r2 * log(well.r3/well.r2))
    h3r = h3
    lambdal_eq = well.lambdaf * nu_a_t

    # fluid inside tubing
    qp = 0.2 * well.q * 2 * (well.f_p[x] * well.rhof[x] * (well.vp ** 2) *
                             (well.md[-1] / (well.dti * 127.094 * 10 ** 6)))

    c1z.append(((well.rhof[x] * well.cf * well.vp) / well.deltaz) / 2)  # Vertical component (North-South)
    c1e.append((2 * well.h1[x] / well.r1) / 2)  # East component
    c1.append(qp / (pi * (well.r1 ** 2)))  # Heat source term
    c1t.append(well.rhof[x] * well.cf / deltat)  # Time component

    # tubing wall
    c2z.append((well.lambdat / (well.deltaz ** 2)) / 2)  # Vertical component (North-South)
    c2e.append((2 * well.r2 * h2 / ((well.r2 ** 2) - (well.r1 ** 2))) / 2)  # East component
    c2w.append((2 * well.r1 * well.h1[x] / ((well.r2 ** 2) - (well.r1 ** 2))) / 2)  # West component
    c2t.append(well.rhot * well.ct / deltat)  # Time component

    if in_section == 1:
        lambda4 = well.lambdar  # Thermal conductivity of the casing (riser in this section)
        lambda5 = well.lambdaw  # Thermal conductivity of the surrounding space (seawater)
        lambda45 = (lambda4 * (well.r4r - well.r3r) + lambda5 * (well.r5 - well.r4r)) / (
                well.r5 - well.r3r)  # Comprehensive Thermal conductivity of the casing (riser) and
                                 # surrounding space (seawater)
        lambda56 = well.lambdaw  # Comprehensive Thermal conductivity of the surrounding space (seawater) and
                               # formation (seawater)
        c4 = well.cr  # Specific Heat Capacity of the casing (riser)
        c5 = well.cw  # Specific Heat Capacity of the surrounding space (seawater)
        rho4 = well.rhor  # Density of the casing (riser)
        rho5 = well.rhow  # Density of the surrounding space (seawater)

        # fluid inside annular
        c3z.append((lambdal_eq / (well.deltaz ** 2)) / 2)  # Vertical component (North-South)
        c3e.append((2 * well.r3 * h3 / ((well.r3 ** 2) - (well.r2 ** 2))) / 2)  # East component
        c3w.append((2 * well.r2 * h2 / ((well.r3 ** 2) - (well.r2 ** 2))) / 2)  # West component
        c3t.append(well.rhof_a[x] * well.cf / deltat)  # Time component

    else:
        # fluid inside annular
        c3z.append((lambdal_eq / (well.deltaz ** 2)) / 2)  # Vertical component (North-South)
        c3e.append((2 * well.r3 * h3r / ((well.r3r ** 2) - (well.r2 ** 2))) / 2)  # East component
        c3w.append((2 * well.r2 * h2 / ((well.r3r ** 2) - (well.r2 ** 2))) / 2)  # West component
        c3t.append(well.rhof_a[x] * well.cf / deltat)  # Time component

    if 1 < in_section < len(sections):

        # calculation for surrounding space
        # thickness
        tcsr = 0
        tcem = 0
        for i in range(len(well.casings) - in_section):
            tcsr += (well.casings[i + 1, 0] - well.casings[i + 1, 1]) / 2
            tcem += (well.casings[i + 1, 1] - well.casings[i, 0]) / 2

            tcem += (well.casings[len(well.casings)-in_section+1, 1] -
                     well.casings[len(well.casings)-in_section, 0]) / 2
        if in_section == 2:
            tcem += (well.dsro - well.casings[-1, 0])
        xcsr = tcsr / (well.r5 - well.r4)  # fraction of surrounding space that is casing
        xcem = tcem / (well.r5 - well.r4)  # fraction of surrounding space that is cement
        xfm = 1 - xcsr - xcem  # fraction of surrounding space that is formation

        # thermal conductivity
        lambdasr = well.lambdac * xcsr + well.lambdacem * xcem + well.lambdafm * xfm
        lambdacsr = (well.lambdac * (well.r4 - well.r3) + lambdasr * (well.r5 - well.r4)) / (well.r5 - well.r3)
        lambdasrfm = (well.lambdac * (well.r5 - well.r4) + lambdasr * (well.rfm - well.r5)) / (well.rfm - well.r4)

        # Specific Heat Capacity
        csr = (well.cc * tcsr + well.ccem * tcem) / (well.r5 - well.r4)
        # Density
        rhosr = xcsr * well.rhoc + xcem * well.rhocem + xfm * well.rhofm

        lambda4 = well.lambdac
        lambda45 = lambdacsr
        lambda5 = lambdasr
        lambda56 = lambdasrfm
        c4 = well.cc  # Specific Heat Capacity of the casing
        c5 = csr  # Specific Heat Capacity of the surrounding space
        rho4 = well.rhoc  # Density of the casing
        rho5 = rhosr  # Density of the surrounding space

    if in_section == len(sections)+1:
        lambda4 = well.lambdafm
        lambda45 = well.lambdafm
        lambda5 = well.lambdafm
        lambda56 = well.lambdafm
        c4 = well.cfm  # Specific Heat Capacity of the casing (formation)
        c5 = well.cfm  # Specific Heat Capacity of the surrounding space (formation)
        rho4 = well.rhofm  # Density of the casing (formation)
        rho5 = well.rhofm  # Density of the surrounding space (formation)

    # first casing wall
    c4z.append((lambda4 / (well.deltaz ** 2)) / 2)
    c4e.append((2 * lambda45 / ((well.r4 ** 2) - (well.r3 ** 2))) / 2)
    c4w.append((2 * well.r3 * h3 / ((well.r4 ** 2) - (well.r3 ** 2))) / 2)
    c4t.append(rho4 * c4 / deltat)

    # surrounding space
    c5z.append((lambda5 / (well.deltaz ** 2)) / 2)
    c5w.append((lambda56 / (well.r5 * (well.r5 - well.r4) * log(well.r5 / well.r4))) / 2)
    c5e.append((lambda56 / (well.r5 * (well.r5 - well.r4) * log(well.rfm / well.r5))) / 2)
    c5t.append(rho5 * c5 / deltat)

hc_1 = [c1z, c1e, c1, c1t]
hc_2 = [c2z, c2e, c2w, c2t]
hc_3 = [c3z, c3e, c3w, c3t]
hc_4 = [c4z, c4e, c4w, c4t]
hc_5 = [c5z, c5e, c5w, c5t]
coefficients = [hc_1, hc_2, hc_3, hc_4, hc_5, cbe, cbt, cbz]

return coefficients
```

D.4. Linearsystem.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/injection/linearsystem.py

```
def define_coef(coefficients, zstep):
    """
    Retrieves respective heat transfer coefficients for certain depth point.
    :param coefficients: list with distribution of heat transfer coefficients
    :param zstep: depth step
    :return: values of heat coefficients for each section at the same depth
    """

    hc_1 = coefficients[0]
    c1z = hc_1[0][zstep]
    c1e = hc_1[1][zstep]
    c1 = hc_1[2][zstep]
    c1t = hc_1[3][zstep]

    hc_2 = coefficients[1]
    c2z = hc_2[0][zstep]
    c2e = hc_2[1][zstep]
    c2w = hc_2[2][zstep]
    c2t = hc_2[3][zstep]

    hc_3 = coefficients[2]
    c3z = hc_3[0][zstep]
    c3e = hc_3[1][zstep]
    c3w = hc_3[2][zstep]
    c3t = hc_3[3][zstep]

    hc_4 = coefficients[3]
    c4z = hc_4[0][zstep]
    c4e = hc_4[1][zstep]
    c4w = hc_4[2][zstep]
    c4t = hc_4[3][zstep]

    hc_5 = coefficients[4]
    c5z = hc_5[0][zstep]
    c5e = hc_5[1][zstep]
    c5w = hc_5[2][zstep]
    c5t = hc_5[3][zstep]

    cbe = coefficients[5]
    cbt = coefficients[6]
    cbz = coefficients[7]

    return c1z, c1e, c1, c1t, c2z, c2e, c2w, c2t, c3z, c3e, c3w, c3t, c4z, c4e, c4w, c4t, c5z, c5e, c5w, c5t, cbe, \
        cbt, cbz
```

```
def temp_calc(well, initcond, heatcoeff):
    """
    Build the penta-diagonal matrix and solve it to get the well temperature distribution.
    :param well: a well object created from the function set_well()
    :param initcond: object with initial temperature profiles
    :param heatcoeff: list with distribution of heat transfer coefficients
    :return: object with final well temperature distribution
    """

    from numpy import zeros, linalg

    Tft = [well.tin]
    Tt = []
    Ta = []
    tc = []
    Tsr = []
    xi = 5

    # Creating vectors N,W,C,E,S,B
    N = []
    W = []
    C = []
    E = []
    S = []
    B = []

    for j in range(well.zstep):
        c1z, c1e, c1, c1t, c2z, c2e, c2w, c2t, c3z, c3e, c3w, c3t, c4z, c4e, c4w, c4t, c5z, c5e, c5w, c5t, cbe, \
        cbt, cbz = define_coef(heatcoeff, j)
        for i in range(xi):

            if i == 0:  # Inside Tubing

                if j == 1:
                    W.append(0)
                    C.append(c1t + c1e + c1z)
                    E.append(-c1e)
                    S.append(0)
                    B.append(c1t * initcond.tfto[j]    # Center(t=0)
                             + c1       # Heat Source
                             + c1e * (initcond.tto[j] - initcond.tfto[j])    # East(t=0)
                             + c1z * (initcond.tfto[j - 1] - initcond.tfto[j])
                             + c1z * well.tin)      # N/S(t=0)

                if 1 < j < well.zstep - 1:
                    N.append(-c1z)
                    W.append(0)
                    C.append(c1t + c1e + c1z)
                    E.append(-c1e)
                    S.append(0)
                    B.append(c1t * initcond.tfto[j]    # Center(t=0)
                             + c1      # Heat Source
                             + c1e * (initcond.tto[j] - initcond.tfto[j])    # East(t=0)
                             + c1z * (initcond.tfto[j - 1] - initcond.tfto[j]))      # N/S(t=0)

                if j == well.zstep - 1:
                    N.append(-c1z)
                    W.append(0)
                    C.append(cbt + c1z)
                    E.append(0)
                    B.append(cbt * initcond.tfto[j]    # Center(t=0)
                             + c1z * (initcond.tfto[j - 1] - initcond.tfto[j]))  # N/S(t=0)

            if i == 1:  # Tubing wall

                if j == 0:
                    C.append(c2t + c2e + c2w + c2z)
                    E.append(-c2e)
                    S.append(-c2z)
                    B.append(c2t * initcond.tto[j]
                             + c2e * (initcond.tao[j] - initcond.tto[j])
                             + c2w * (initcond.tfto[j] - initcond.tto[j])
                             + c2z * (initcond.tto[j + 1] - initcond.tto[j])
                             + c2w * well.tin)

                if 0 < j < well.zstep - 1:
                    N.append(-c2z)
                    W.append(-c2w)
                    C.append(c2t + c2e + c2w + 2 * c2z)
                    E.append(-c2e)
                    S.append(-c2z)
                    B.append(c2t * initcond.tto[j]
                             + c2e * (initcond.tao[j] - initcond.tto[j])
                             + c2w * (initcond.tfto[j] - initcond.tto[j])
                             + c2z * (initcond.tto[j + 1] - initcond.tto[j])
                             + c2z * (initcond.tto[j - 1] - initcond.tto[j]))

                if j == well.zstep - 1:
                    N.append(-c2z)
                    W.append(0)
                    C.append(c2t + c2z)
                    E.append(0)
                    B.append(c2t * initcond.tto[j]
                             + c2z * (initcond.tto[j - 1] - initcond.tto[j]))

            if i == 2:  # Annular

                if j == 0:
                    W.append(-c3w)
                    C.append(c3t + c3e + c3w + c3z)
                    E.append(-c3e)
                    S.append(-c3z)
                    B.append(c3t * initcond.tao[j]
                             + c3e * (initcond.tco[j] - initcond.tao[j])
                             + c3w * (initcond.tto[j] - initcond.tao[j])
                             + c3z * (initcond.tao[j + 1] - initcond.tao[j]))

                if 0 < j < well.zstep - 1:
                    N.append(-c3z)
                    W.append(-c3w)
                    C.append(c3t + c3e + c3w + 2 * c3z)
                    E.append(-c3e)
                    S.append(-c3z)
                    B.append(c3t * initcond.tao[j]
                             + c3e * (initcond.tco[j] - initcond.tao[j])
                             + c3w * (initcond.tto[j] - initcond.tao[j])
                             + c3z * (initcond.tao[j + 1] - initcond.tao[j])
                             + c3z * (initcond.tao[j - 1] - initcond.tao[j]))

                if j == well.zstep - 1:
                    N.append(-c3z)
                    W.append(0)
                    C.append(c3t + c3e + c3z)
                    E.append(-c3e)
                    B.append(c3t * initcond.tao[j]
                             + c3e * (initcond.tco[j] - initcond.tao[j])
                             + c3z * (initcond.tao[j - 1] - initcond.tao[j]))

            if i == 3:  # Casing

                if j == 0:
                    W.append(-c4w)
                    C.append(c4t + c4e + c4w + c4z)
                    E.append(-c4e)
                    S.append(-c4z)
                    B.append(c4t * initcond.tco[j]   # Center(t=0)
                             + c4e * (initcond.tsro[j] - initcond.tco[j])  # East(t=0)
                             + c4w * (initcond.tao[j] - initcond.tco[j])  # West(t=0)
                             + c4z * (initcond.tco[j + 1] - initcond.tco[j]))   # N/S(t=0)

                if 0 < j < well.zstep - 1:
                    N.append(-c4z)
                    W.append(-c4w)
                    C.append(c4t + c4e + c4w + 2 * c4z)
                    E.append(-c4e)
                    S.append(-c4z)
                    B.append(c4t * initcond.tco[j]    # Center(t=0)
                             + c4e * (initcond.tsro[j] - initcond.tco[j])     # East(t=0)
                             + c4w * (initcond.tao[j] - initcond.tco[j])      # West(t=0)
```

```
                + c4z * (initcond.tco[j + 1] - 2 * initcond.tco[j] + initcond.tco[j - 1])) # N/S(t=0)

        if j == well.zstep - 1:
            N.append(-c4z)
            W.append(-c4w)
            C.append(c4t + c4e + c4w + c4z)
            E.append(-c4e)
            B.append(c4t * initcond.tco[j]    # Center(t=0)
                + c4e * (initcond.tsro[j] - initcond.tco[j])    # East(t=0)
                + c4w * (initcond.tao[j] - initcond.tco[j])     # West(t=0)
                + c4z * (initcond.tco[j - 1] - initcond.tco[j]))      # N/S(t=0))

    if i == 4:  # Surrounding Space

        if j == 0:
            W.append(-c5w)
            C.append(c5w + c5z + c5e + c5t)
            E.append(0)
            S.append(-c5z)
            B.append(c5w * (initcond.tco[j] - initcond.tsro[j])
                + c5z * (initcond.tsro[j + 1] - initcond.tsro[j])
                + c5e * initcond.tsro[j])

        if 0 < j < well.zstep - 1:
            N.append(-c5z)
            W.append(-c5w)
            C.append(c5w + c5e + 2 * c5z + c5t)
            E.append(0)
            S.append(-c5z)
            B.append(c5w * (initcond.tco[j] - initcond.tsro[j])
                + c5z * (initcond.tsro[j + 1] - initcond.tsro[j])
                + c5z * (initcond.tsro[j - 1] - initcond.tsro[j])
                + c5e * initcond.tsro[j])

        if j == well.zstep - 1:
            N.append(-c5z)
            W.append(-c5w)
            C.append(c5w + c5e + c5z + c5t)
            B.append(c5w * (initcond.tco[j] - initcond.tsro[j])
                + c5z * (initcond.tsro[j - 1] - initcond.tsro[j])
                + c5t * initcond.tsro[j]
                + c5e * initcond.tsro[j])

#LINEARSYSTEM
# Creating pentadiagonal matrix
A = zeros((xi * well.zstep - 1, xi * well.zstep - 1))

# Filling up Pentadiagonal Matrix A
lenC = xi * well.zstep - 1
lenN = lenC - xi
```

```
lenW = lenC - 1
lenE = lenC - 1
lenS = lenC - xi

for it in range(lenC):  # Inserting list C
    A[it, it] = C[it]
for it in range(lenE):  # Inserting list E
    A[it, it + 1] = E[it]
for it in range(lenW):  # Inserting list W
    A[it + 1, it] = W[it]
for it in range(lenN):  # Inserting list N
    A[it + xi, it] = N[it]
for it in range(lenS):  # Inserting list S
    A[it, it + xi] = S[it]

Temp = linalg.solve(A, B)

for x in range(well.zstep):
    Tt.append(Temp[5 * x])
    if x < well.zstep-1:
        Tft.append(Temp[5 * x + 4])
        Ta.append(Temp[5 * x + 1])
    if x == well.zstep - 1:
        Ta.append(Tft[-1])
    tc.append(Temp[5 * x + 2])
    Tsr.append(Temp[5 * x + 3])


t3 = tc.copy()

tr = tc[:well.riser] + [None] * (well.zstep - well.riser)
for x in range(well.riser):
    tc[x] = None

csgs_reach = int(well.casings[0, 2] / well.deltaz)  # final depth still covered with casing(s)

Toh = [None] * csgs_reach + tc[csgs_reach:]
for x in range(csgs_reach, well.zstep):
    tc[x] = None

class TempCalc(object):
    def __init__(self):
        self.tft = Tft
        self.tt = Tt
        self.ta = Ta
        self.t3 = t3
        self.tc = tc
        self.tr = tr
        self.tsr = Tsr
        self.toh = Toh
        self.csgs_reach = csgs_reach

    return TempCalc()
```

D.5. Fluid.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/injection/fluid.py

```
def initial_density(well, initcond, section='tubing'):
    """
    Function to calculate the density profile for the first time step
    :param section: 'tubing' or 'annular'
    :param well: a well object created from the function set_well()
    :param initcond: a initial conditions object with the formation temperature profile
    :return: the density profile and the initial density at surface conditions
    """

    if section == 'tubing':
        beta = well.beta
        alpha = well.alpha
        rho = well.rhof
    if section == 'annular':
        beta = well.beta_a
        alpha = well.alpha_a
        rho = well.rhof_a

    rhof_initial = rho
    pressure = [rho * 9.81 * i for i in well.tvd]
    rhof = [rhof_initial * (1 + (x - 10 ** 5) / beta - alpha * (y - well.ts)) for x, y in
            zip(pressure, initcond.tfto)]
    pressure = [x * 9.81 * y for x, y in zip(rhof, well.tvd)]
    rhof = [rhof_initial * (1 + (x - 10 ** 5) / beta - alpha * (y - well.ts)) for x, y in
            zip(pressure, initcond.tfto)]

    return rhof, rhof_initial


def calc_density(well, initcond, rhof_initial, section='tubing'):
    """
    Function to calculate the density profile
    :param section: 'tubing' or 'annular'
    :param well: a well object created from the function set_well()
    :param initcond: a initial conditions object with the formation temperature profile
    :param rhof_initial: initial density at surface conditions
    :param flow: boolean to define if the section is flowing
    :return: density profile
    """

    if section == 'tubing':
        beta = well.beta
        alpha = well.alpha
        flow = True
        temp = initcond.tfto
    rho = well.rhof
if section == 'annular':
    beta = well.beta_a
    alpha = well.alpha_a
    flow = False
    temp = initcond.tao
    rho = well.rhof_a

pressure_h = [x * 9.81 * y for x, y in zip(rho, well.tvd)]

if flow:
    pressure_f = [x * (well.md[-1] / well.dti) * (1/2) * y * well.vp **2 for x, y in zip(well.f_p, rho)]
else:
    pressure_f = [0] * len(well.md)

pressure = [x + y for x, y in zip(pressure_h, pressure_f)]

rhof = [rhof_initial * (1 + (x - 10 ** 5) / beta - alpha * (y - well.ts)) for x, y in
        zip(pressure, temp)]

return rhof
```

D.6. Main.py

https://github.com/pro-well-plan/pwptemp/blob/master/pwptemp/injection/main.py

```
def temp_time(n, well, log=True, units='metric', time_delta=None):
    """
    Function to calculate the well temperature distribution during certain production time (n)
    :param n: production time, hours
    :param well: a well object created with the function set_well() from input.py
    :param log: save distributions between initial time and circulation time n (each 1 hour)
    :param units: system of units ('metric' or 'english')
    :param time_delta: duration of each time step (seconds)
    :return: a well temperature distribution object
    """
    from .initcond import init_cond
    from .heatcoefficients import heat_coef
    from .linearsystem import temp_calc
    from .plot import profile
    from math import log, nan
    import numpy as np
    # Simulation main parameters
    time = n  # circulating time, h
    tcirc = time * 3600  # circulating time, s
    deltat = 60 * time
    if type(time_delta) == int:
        deltat = time_delta
    tstep = int(tcirc / deltat)
    ic = init_cond(well)
    tfm = ic.tfm
    tt = ic.tto
    t3 = ic.tco

    well = well.define_density(ic, cond=0)

    hc = heat_coef(well, deltat, tt, t3)
    temp = temp_calc(well, ic, hc)
    temp.tft = temp.tt = temp.ta = temp.t3 = temp.tsr = tfm
    for x in range(len(tfm)):
        if temp.tc[x] != nan:
            temp.tc[x] = tfm[x]
        if temp.tr[x] != nan:
            temp.tr[x] = tfm[x]
        if temp.toh[x] != nan:
            temp.toh[x] = tfm[x]

    temp_initial = temp
    temp_initial.tft = ic.tfm
    temp_initial.tt = ic.tfm
    temp_initial.ta = ic.tfm
```

```
temp_log = [temp_initial, temp]
time_log = [0, deltat / 3600]

for x in range(tstep-1):
    well = well.define_density(ic, cond=1)

    ic.tfto = temp.tft
    ic.tto = temp.tt
    ic.tao = temp.ta
    ic.tco = temp.t3
    ic.tsr = temp.tsr
    hc_new = heat_coef(well, deltat, ic.tto, ic.tco)
    temp = temp_calc(well, ic, hc_new)

    if units == 'english':
        temp.tft_output = [(i / (5 / 9) + 32) for i in temp.tft]
        temp.tt_output = [(i / (5 / 9) + 32) for i in temp.tt]
        temp.ta_output = [(i / (5 / 9) + 32) for i in temp.ta]
        temp.tc_output = [(i / (5 / 9) + 32) for i in temp.tc if type(i) == np.float64]
        temp.tr_output = [(i / (5 / 9) + 32) for i in temp.tr if type(i) == np.float64]
        temp.tsr_output = [(i / (5 / 9) + 32) for i in temp.tsr]
        temp.md_output = [i * 3.28 for i in well.md]

    if log:
        temp_log.append(temp)
        time_log.append(time_log[-1] + time_log[1])

if units == 'english':
    temp.tft = temp.tft_output
    temp.tt = temp.tt_output
    temp.ta = temp.ta_output
    temp.tc = temp.tc_output
    temp.tr = temp.tr_output
    temp.sr = temp.tsr_output
    temp.md = temp.md_output
    tfm = [(i / (5 / 9) + 32) for i in tfm]

class TempDist(object):
    def __init__(self):
        self.tft = temp.tft
        self.tt = temp.tt
        self.ta = temp.ta
        self.tc = temp.tc
        self.tr = temp.tr
        self.tsr = temp.tsr
        self.tfm = tfm
        self.time = time
        self.md = well.md
        self.riser = well.riser
```

```
            self.deltat = deltat
            self.csgs_reach = temp.csgs_reach
            if log:
                self.temp_log = temp_log
                self.time_log = time_log

        def well(self):
            return well

        def plot(self, tft=True, tt=False, ta=True, tc=False, tr=False, sr=False):
            profile(self, tft, tt, ta, tc, tr, sr, units)

        def behavior(self):
            temp_behavior_injection = temp_behavior(self)
            return temp_behavior_injection

        def plot_multi(self, tft=True, ta=False, tr=False, tc=False, tfm=False, tsr=False):
            plot_multitime(self, tft, ta, tr, tc, tfm, tsr)

    return TempDist()


def temp_behavior(temp_dist):

    tft = [x.tft for x in temp_dist.temp_log]

    tbot = []

    for n in range(len(tft)):
        tbot.append(tft[n][-1])

    class Behavior(object):
        def __init__(self):
            self.finaltime = temp_dist.time
            self.tbot = tbot
            self.tfm = temp_dist.tfm
            self.time = temp_dist.time_log

        def plot(self):
            from .plot import behavior
            behavior(self)

    return Behavior()


def plot_multitime(temp_dist, tft=True, ta=False, tr=False, tc=False, tfm=False, tsr=False):
    from .plot import profile_multitime

    values = temp_dist.temp_log
    times = [x for x in temp_dist.time_log]
    profile_multitime(temp_dist, values, times, tft=tft, ta=ta, tr=tr, tc=tc, tfm=tfm, tsr=tsr)


def temp(n, mdt=3000, casings=[], wellpath_data=[], d_openhole=0.216, grid_length=50, profile='V', build_angle=1,
kop=0,
         eob=0, sod=0, eod=0, kop2=0, eob2=0, change_input={}, log=False, units='metric', time_delta=None):
    """
    Main function to calculate the well temperature distribution during production operation. This function allows to
    set the wellpath and different parameters involved.
    :param n: production time, hours
    :param mdt: measured depth of target, m
    :param casings: list of dictionaries with casings characteristics (od, id and depth)
    :param wellpath_data: load own wellpath as a list
    :param d_openhole: diameter of open hole section, m
    :param grid_length: number of cells through depth
    :param profile: type of well to generate. Vertical ('V'), S-type ('S'), J-type ('J') and Horizontal ('H1' or 'H2')
    :param build_angle: build angle, °
    :param kop: kick-off point, m
    :param eob: end of build, m
    :param sod: start of drop, m
    :param eod: end of drop, m
    :param kop2: kick-off point 2, m
    :param eob2: end of build 2, m
    :param change_input: dictionary with parameters to set.
    :param log: save distributions between initial time and circulation time n (each 1 hour)
    :param units: system of units ('metric' or 'english')
    :param time_delta: duration of each time step (seconds)
    :return: a well temperature distribution object
    """
    from .input import data, set_well
    from .. import wellpath
    tdata = data(casings, d_openhole, units)
    for x in change_input:   # changing default values
        if x in tdata:
            tdata[x] = change_input[x]
        else:
            raise TypeError('%s is not a parameter' % x)
    if len(wellpath_data) == 0:
        depths = wellpath.get(mdt, grid_length, profile, build_angle, kop, eob, sod, eod, kop2, eob2, units)
    else:
        depths = wellpath.load(wellpath_data, grid_length, units)
    well = set_well(tdata, depths, units)
    temp_distribution = temp_time(n, well, log, units, time_delta)

    return temp_distribution


def input_info(about='all'):
```

```
from .input import info
info(about)
```

Appendix E Temperature Profile – Prediction Model

E.1. Drilling Prediction Model

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pwptemp.drilling as ptd
import lightgbm as lgb
from sklearn.metrics import mean_squared_error, r2_score #for checking the model accuracy
from sklearn.model_selection import cross_val_score, train_test_split

# Loading the dataset:
drilling_data = pd.read_csv('drilling_cases.csv')

# The model will actually predict the change in temperature from the initial value (formation temperature). Therefore,
# let's create the column 'Tdsi_change' to use it as target.
case1 = drilling_data[['Time','Flow_rate','Density','Depth','Tdsi','Tfm']].copy()
case1['Tdsi_change'] = case1['Tdsi'] - case1['Tfm']
case1.drop(['Tdsi','Tfm'], axis=1, inplace=True)

# Split data: train and test datasets
train, test = train_test_split(case1, test_size=0.3, shuffle = True)
train_x = train.drop(['Tdsi_change'], axis=1)
train_y = train.Tdsi_change.values
test_x = test.drop(['Tdsi_change'], axis=1)
test_y = test.Tdsi_change.values
print(len(train_y), ' rows for training and ', len(test_y), ' for testing')

# LightGBM is a gradient boosting framework that uses tree based learning algorithms. Let's use it to generate the model
# and then calculate the mean squared error and R2:
model_lgbm = lgb.LGBMRegressor(objective='regression', num_leaves=5,
                        learning_rate=0.05, n_estimators=400,
                        max_bin = 55,
                        min_data_in_leaf = 2, min_sum_hessian_in_leaf = 2)

model_lgbm.fit(train_x, train_y)
prediction = model_lgbm.predict(test_x.values)
print('Mean Squared Error: {:.4f}'.format(mean_squared_error(test_y, prediction, squared=False)))
r2_lgbm = round(r2_score(test_y, prediction), 4)
print('R-squared: {:.4f}'.format(r2_lgbm))
score = cross_val_score(model_lgbm, train.values, train_y, cv = 5)
print('Cross Validation',score)
print("LGBM score: {:.4f} ({:.4f})\n" .format(score.mean(), score.std()))

# Plotting the results from predictions made using the test dataset:
lgbm_label = 'LGBM, R2: {:.4f}'.format(r2_lgbm)
```

```
plt.plot(list(range(20)), prediction[:20], 'b-', label=lgbm_label)
plt.plot(list(range(20)), test_y[:20], 'go', label='True Data')
plt.xlabel('Data point')
plt.ylabel('Tdsi_change, °C')
plt.title('True Data vs Prediction')
plt.legend()
plt.show()

# Comparing prediction with a simulation
# Get the results from simulating a drilling process at 3 hours of operation. By default pwptemp set a target depth of
# 3000 m, flow rate of 794.933 lpm and mud density of 1.198 sg. Assuming surface temperature of 20°C and inlet fluid
# temperature of 25°C.
res = ptd.temp(3, change_input={'ts':20, 'tin':25})
simulated = res.tdsi

# Now get the predicted results:
res_predict = model_lgbm.predict([[3,794.933,1.198,x] for x in range(0,3050,50)])
res_predict = [res_predict[x] + res.tfm[x] for x in range(len(res.tdsi))]
predicted = res_predict

# Plotting both results:
md = list(range(0,3050,50))
R2 = r2_score(simulated, predicted)
plt.plot(simulated, md, c='r', label='Simulated') # Temp. inside Drillpipe vs Depth
plt.plot(predicted, md, c='b', label='Predicted, R2: {:.4f}'.format(R2)) # Temp. inside Drillpipe vs Depth
plt.xlabel('Temperature, °C')
plt.ylabel('Depth, m')
plt.ylim(0, md[-1])     # bottom and top limits
plt.ylim(plt.ylim()[::-1])  # reversing y axis
plt.legend()  # applying the legend
plt.grid()
plt.show()

# Including a Correction Factor
# Let's use 4 cases:
# 1. Base case: t = 1h, q = 794.933lpm, rhof = 1.198 sg
# 2. Time case: t = 20h, q = 794.933lpm, rhof = 1.198 sg
# 3. Flow rate case: t = 1h, q = 2500lpm, rhof = 1.198 sg
# 4. Density case: t = 1h, q = 794.933lpm, rhof = 1.85 sg

# Set the parameters
t1 = 1
t2 = 20
q1 = 794.933
q2 = 2500
rhof1 = 1.198
rhof2 = 1.85

# Base case
res_sim1 = ptd.temp(t1, change_input={'ts':20, 'tin':25})
```

```
# Time case
res_sim2 = ptd.temp(t2, change_input={'ts':20, 'tin':25})
# Flow rate case
res_sim3 = ptd.temp(t1, change_input={'q':q2, 'ts':20, 'tin':25})
# Density change
res_sim4 = ptd.temp(t1, change_input={'rhof':rhof2, 'ts':20, 'tin':25})

# Get prediction results:
# Base case
res_pred1 = model_lgbm.predict([[t1,q1,rhof1,x] for x in range(0,3050,50)])
res_pred1 = [res_pred1[x] + res_sim1.tfm[x] for x in range(len(res_sim1.tdsi))]
# Time case
res_pred2 = model_lgbm.predict([[t2,q1,rhof1,x] for x in range(0,3050,50)])
res_pred2 = [res_pred2[x] + res_sim2.tfm[x] for x in range(len(res_sim2.tdsi))]
# Flow rate case
res_pred3 = model_lgbm.predict([[t1,q2,rhof1,x] for x in range(0,3050,50)])
res_pred3 = [res_pred3[x] + res_sim3.tfm[x] for x in range(len(res_sim3.tdsi))]
# Density change
res_pred4 = model_lgbm.predict([[t1,q1,rhof2,x] for x in range(0,3050,50)])
res_pred4 = [res_pred4[x] + res_sim4.tfm[x] for x in range(len(res_sim4.tdsi))]

# Using the cases below we can create a correction profile, just check the difference in values for each case between
# predicted and simulated, and then calculate the coefficients for the correction function:
dif1 = [res_pred1[x]-res_sim1.tdsi[x] for x in range(len(res_sim1.tdsi))]
dif2 = [res_pred2[x]-res_sim2.tdsi[x] for x in range(len(res_sim2.tdsi))]
dif3 = [res_pred3[x]-res_sim3.tdsi[x] for x in range(len(res_sim3.tdsi))]
dif4 = [res_pred4[x]-res_sim4.tdsi[x] for x in range(len(res_sim4.tdsi))]

def get_correction(t, q, rhof):
    m_time = [((dif2[x] - dif1[x]) / (t2 - t1)) for x in range(len(md))]
    m_q = [((dif3[x] - dif1[x]) / (q2 - q1)) for x in range(len(md))]
    m_rhof = [((dif4[x] - dif1[x]) / (rhof2 - rhof1)) for x in range(len(md))]

    correction = [dif1[x] + m_time[x] * (t - t1) + m_q[x] * (q - q1) + m_rhof[x] * (rhof - rhof1) for x in
            range(len(dif1))]

    return correction

# Results
# Now that we have already our correction function, let's use it with the prediction model.
# Set new parameters for circulation time, flow rate and density:
t = 16
q = 1800
rhof = 1.3

# Generate simulation a prediction to do the comparison. Additionally we can check the execution time involved for each
# one.

# Simulation
%%time
```

```
res_sim5 = ptd.temp(t, change_input={'q':q, 'rhof':rhof, 'ts':20, 'tin':25})

# Prediction
%%time
res_pred5 = model_lgbm.predict([[t,q,rhof,x] for x in range(0,3050,50)])
res_pred5 = [res_pred5[x] + res_sim5.tfm[x] for x in range(len(md))]
correction = get_correction(t, q, rhof)
res_pred5_corrected = [res_pred5[x] - correction[x] for x in range(len(correction))]

# Plotting Simulated vs Predicted
name = [res_sim5, res_pred5, res_pred5_corrected]
md = list(range(0,3050,50))
r2_2 = r2_score(res_sim5.tdsi, res_pred5_corrected)
plt.plot(name[0].tdsi, md, c='r', label='Simulated') # Temp. inside Drillpipe vs Depth
plt.plot(name[2], md, c='b', label='Predicted, R2: {:.4f}'.format(r2_2)) # Temp. inside Drillpipe vs Depth
plt.xlabel('Temperature, °C')
plt.ylabel('Depth, m')
plt.ylim(0, md[-1])     # bottom and top limits
plt.ylim(plt.ylim()[::-1])  # reversing y axis
plt.legend()  # applying the legend
plt.grid()
plt.show()
```

E.2. Production Prediction Model

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pwptemp.production as ptp
import lightgbm as lgb
from sklearn.metrics import mean_squared_error, r2_score #for checking the model accuracy
from sklearn.model_selection import cross_val_score, train_test_split

# Loading the dataset:
production_data = pd.read_csv('production_cases.csv')

# The model will actually predict the change in temperature from the initial value (formation temperature). Therefore,
# let's create the column 'tft_change' to use it as target.
case1 = production_data[['Time','Flow_rate','Density','Depth','Tft','Tfm']].copy()
case1['Tft_change'] = case1['Tft'] - case1['Tfm']
case1.drop(['Tft','Tfm'], axis=1, inplace=True)

# Split data: train and test datasets
train, test = train_test_split(case1, test_size=0.3, shuffle = True)
train_x = train.drop(['Tft_change'], axis=1)
train_y = train.Tft_change.values
test_x = test.drop(['Tft_change'], axis=1)
test_y = test.Tft_change.values
print(len(train_y), ' rows for training and ', len(test_y), ' for testing')

# LightGBM is a gradient boosting framework that uses tree based learning algorithms. Let's use it to generate the model
# and then calculate the mean squared error and R2:
model_lgbm = lgb.LGBMRegressor(objective='regression', num_leaves=5,
                    learning_rate=0.05, n_estimators=400,
                    max_bin = 55,
                    min_data_in_leaf = 2, min_sum_hessian_in_leaf = 2)

model_lgbm.fit(train_x, train_y)
prediction = model_lgbm.predict(test_x.values)
print('Mean Squared Error: {:.4f}'.format(mean_squared_error(test_y, prediction, squared=False)))
r2_lgbm = round(r2_score(test_y, prediction), 4)
print('R-squared: {:.4f}'.format(r2_lgbm))
score = cross_val_score(model_lgbm, train.values, train_y, cv = 5)
print('Cross Validation',score)
print("LGBM score: {:.4f} ({:.4f})\n" .format(score.mean(), score.std()))

# Plotting the results from predictions made using the test dataset:
lgbm_label = 'LGBM, R2: {:.4f}'.format(r2_lgbm)
plt.plot(list(range(20)), prediction[:20], 'b-', label=lgbm_label)
plt.plot(list(range(20)), test_y[:20], 'go', label='True Data')
plt.xlabel('Data point')
plt.ylabel('Tft_change, °C')
plt.title('Simulated Data vs Prediction')
plt.legend()
plt.show()

# Comparing prediction with a simulation
# Get the results from simulating a production process at 9 hours of operation. By default pwptemp set a target depth
# of 3000 m and flow rate of 2000 m3/d. Let's assume an oil of 800 kg/m3.
res = ptp.temp(9, change_input={'rhof':0.8, 'q':2000, 'ts':20})
simulated = res.tft

# Now get the predicted results:
res_predict = model_lgbm.predict([[9,2000,0.8,x] for x in range(0,3050,50)])
res_predict = [res_predict[x] + res.tfm[x] for x in range(len(res.tft))]
predicted = res_predict

# Plotting both results:
md = list(range(0,3050,50))
R2 = r2_score(simulated, predicted)
plt.plot(simulated, md, c='r', label='Simulated') # Temp. inside Tubing vs Depth
plt.plot(predicted, md, c='b', label='Predicted, R2: {:.4f}'.format(R2)) # Temp. inside Tubing vs Depth
plt.xlabel('Temperature, °C')
plt.ylabel('Depth, m')
plt.ylim(0, md[-1])     # bottom and top limits
plt.ylim(plt.ylim()[::-1])  # reversing y axis
plt.legend()  # applying the legend
plt.grid()
plt.show()

# Including a Correction Factor
# Let's use 4 cases:
# 1. Base case: t = 1h, q = 2000m3/d, rhof = 0.8 sg
# 2. Time case: t = 60h, q = 2000m3/d, rhof = 0.8 sg
# 3. Flow rate case: t = 1h, q = 4500m3/d, rhof = 0.8 sg
# 4. Density case: t = 1h, q = 2000m3/d, rhof = 0.95 sg

# Set the parameters
t1 = 10
t2 = 60
q1 = 2000
q2 = 4500
rhof1 = 0.8
rhof2 = 0.95

# Base case
res_sim1 = ptp.temp(t1, change_input={'rhof':0.8, 'q':2000, 'ts':20})
# Time case
res_sim2 = ptp.temp(t2, change_input={'rhof':0.8, 'q':2000, 'ts':20})
# Flow rate case
res_sim3 = ptp.temp(t1, change_input={'rhof':0.8, 'q':q2, 'ts':20})
# Density change
```

```
res_sim4 = ptp.temp(t1, change_input={'rhof':rhof2, 'q':2000, 'ts':20})

# Get prediction results:
# Base case
res_pred1 = model_lgbm.predict([[t1,q1,rhof1,x] for x in range(0,3050,50)])
res_pred1 = [res_pred1[x] + res_sim1.tfm[x] for x in range(len(res_sim1.tft))]
# Time case
res_pred2 = model_lgbm.predict([[t2,q1,rhof1,x] for x in range(0,3050,50)])
res_pred2 = [res_pred2[x] + res_sim2.tfm[x] for x in range(len(res_sim2.tft))]
# Flow rate case
res_pred3 = model_lgbm.predict([[t1,q2,rhof1,x] for x in range(0,3050,50)])
res_pred3 = [res_pred3[x] + res_sim3.tfm[x] for x in range(len(res_sim3.tft))]
# Density change
res_pred4 = model_lgbm.predict([[t1,q1,rhof2,x] for x in range(0,3050,50)])
res_pred4 = [res_pred4[x] + res_sim4.tfm[x] for x in range(len(res_sim4.tft))]

# Using the cases below we can create a correction profile, just check the difference in values for each case between
# predicted and simulated, and then calculate the coefficients for the correction function:
dif1 = [res_pred1[x]-res_sim1.tft[x] for x in range(len(res_sim1.tft))]
dif2 = [res_pred2[x]-res_sim2.tft[x] for x in range(len(res_sim2.tft))]
dif3 = [res_pred3[x]-res_sim3.tft[x] for x in range(len(res_sim3.tft))]
dif4 = [res_pred4[x]-res_sim4.tft[x] for x in range(len(res_sim4.tft))]

def get_correction(t, q, rhof):
    m_time = [((dif2[x] - dif1[x]) / (t2 - t1)) for x in range(len(md))]
    m_q = [((dif3[x] - dif1[x]) / (q2 - q1)) for x in range(len(md))]
    m_rhof = [((dif4[x] - dif1[x]) / (rhof2 - rhof1)) for x in range(len(md))]

    correction = [dif1[x] + m_time[x] * (t - t1) + m_q[x] * (q - q1) + m_rhof[x] * (rhof - rhof1) for x in
            range(len(dif1))]

    return correction

# Results
# Now that we have already our correction function, let's use it with the prediction model.
# Set new parameters for circulation time, flow rate and density:
t = 12
q = 2800
rhof = 0.88

# Generate simulation a prediction to do the comparison. Additionally we can check the execution time involved for each
# one.

# Simulation
%%time
res_sim5 = ptp.temp(t, change_input={'q':q, 'rhof':rhof, 'ts':20})

# Prediction
%%time
res_pred5 = model_lgbm.predict([[t,q,rhof,x] for x in range(0,3050,50)])
```

```
res_pred5 = [res_pred5[x] + res_sim5.tfm[x] for x in range(len(md))]
correction = get_correction(t, q, rhof)
res_pred5_corrected = [res_pred5[x] - correction[x] for x in range(len(correction))]

# Plotting Simulated vs Predicted
name = [res_sim5, res_pred5, res_pred5_corrected]
md = list(range(0,3050,50))
r2_2 = r2_score(res_sim5.tft, res_pred5_corrected)
plt.plot(name[0].tft, md, c='r', label='Simulated') # Temp. inside Tubing vs Depth
plt.plot(name[2], md, c='b', label='Predicted, R2: {:.4f}'.format(r2_2)) # Temp. inside Tubing vs Depth
plt.xlabel('Temperature, °C')
plt.ylabel('Depth, m')
plt.ylim(0, md[-1])     # bottom and top limits
plt.ylim(plt.ylim()[::-1])  # reversing y axis
plt.legend()  # applying the legend
plt.grid()
plt.show()
```

E.3. Injection Prediction Model

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pwptemp.injection as pti
import lightgbm as lgb
from sklearn.metrics import mean_squared_error, r2_score #for checking the model accuracy
from sklearn.model_selection import cross_val_score, train_test_split

# Loading the dataset:
injection_data = pd.read_csv('injection_cases.csv')

# The model will actually predict the change in temperature from the initial value (formation temperature). Therefore,
# let's create the column 'tft_change' to use it as target.
case1 = injection_data[['Time','Flow_rate','Density','Depth','Tft','Tfm']].copy()
case1['Tft_change'] = case1['Tft'] - case1['Tfm']
case1.drop(['Tft','Tfm'], axis=1, inplace=True)

# Split data: train and test datasets
train, test = train_test_split(case1, test_size=0.3, shuffle = True)
train_x = train.drop(['Tft_change'], axis=1)
train_y = train.Tft_change.values
test_x = test.drop(['Tft_change'], axis=1)
test_y = test.Tft_change.values
print(len(train_y), ' rows for training and ', len(test_y), ' for testing')

# LightGBM is a gradient boosting framework that uses tree based learning algorithms. Let's use it to generate the model
# and then calculate the mean squared error and R2:
model_lgbm = lgb.LGBMRegressor(objective='regression', num_leaves=5,
                              learning_rate=0.05, n_estimators=400,
                              max_bin = 55,
                              min_data_in_leaf = 2, min_sum_hessian_in_leaf = 2)

model_lgbm.fit(train_x, train_y)
prediction = model_lgbm.predict(test_x.values)
print('Mean Squared Error: {:.4f}'.format(mean_squared_error(test_y, prediction, squared=False)))
r2_lgbm = round(r2_score(test_y, prediction), 4)
print('R-squared: {:.4f}'.format(r2_lgbm))
score = cross_val_score(model_lgbm, train.values, train_y, cv = 5)
print('Cross Validation',score)
print("LGBM score: {:.4f} ({:.4f})\n" .format(score.mean(), score.std()))

# Plotting the results from predictions made using the test dataset:
lgbm_label = 'LGBM, R2: {:.4f}'.format(r2_lgbm)
plt.plot(list(range(20)), prediction[:20], 'b-', label=lgbm_label)
plt.plot(list(range(20)), test_y[:20], 'go', label='True Data')
plt.xlabel('Data point')
plt.ylabel('Tft_change, °C')
plt.title('Simulated Data vs Prediction')
plt.legend()
plt.show()

# Comparing prediction with a simulation
# Get the results from simulating an injection process at 23 hours of operation. By default pwptemp set a target depth
# of 3000 m, flow rate of 144 m3/d and mud density of 1.198 sg. Assuming surface temperature of 20°C and inlet fluid
# temperature of 25°C.
res = pti.temp(23, change_input={'ts':20, 'tin':25})
simulated = res.tft

# Now get the predicted results:
res_predict = model_lgbm.predict([[23,144,1.198,x] for x in range(0,3050,50)])
res_predict = [res_predict[x] + res.tfm[x] for x in range(len(res.tft))]
predicted = res_predict

# Plotting both results:
md = list(range(0,3050,50))
R2 = r2_score(simulated, predicted)
plt.plot(simulated, md, c='r', label='Simulated') # Temp. inside Tubing vs Depth
plt.plot(predicted, md, c='b', label='Predicted, R2: {:.4f}'.format(R2)) # Temp. inside Tubing vs Depth
plt.xlabel('Temperature, °C')
plt.ylabel('Depth, m')
plt.ylim(0, md[-1])    # bottom and top limits
plt.ylim(plt.ylim()[::-1])  # reversing y axis
plt.legend()  # applying the legend
plt.grid()
plt.show()

# Including a Correction Factor
# Let's use 4 cases:
# 1. Base case: t = 1h, q = 144m3/d, rhof = 1.198 sg
# 2. Time case: t = 30h, q = 144m3/d, rhof = 1.198 sg
# 3. Flow rate case: t = 1h, q = 350m3/d, rhof = 1.198 sg
# 4. Density case: t = 1h, q = 144m3/d, rhof = 1.3 sg

# Set the parameters
t1 = 1
t2 = 30
q1 = 144
q2 = 350
rhof1 = 1.198
rhof2 = 1.3

# Base case
res_sim1 = pti.temp(t1, change_input={'ts':20, 'tin':25})
# Time case
res_sim2 = pti.temp(t2, change_input={'ts':20, 'tin':25})
# Flow rate case
res_sim3 = pti.temp(t1, change_input={'q':q2, 'ts':20, 'tin':25})
```

```
# Density change
res_sim4 = pti.temp(t1, change_input={'rhof':rhof2, 'ts':20, 'tin':25})

# Get prediction results:
# Base case
res_pred1 = model_lgbm.predict([[t1,q1,rhof1,x] for x in range(0,3050,50)])
res_pred1 = [res_pred1[x] + res_sim1.tfm[x] for x in range(len(res_sim1.tft))]
# Time case
res_pred2 = model_lgbm.predict([[t2,q1,rhof1,x] for x in range(0,3050,50)])
res_pred2 = [res_pred2[x] + res_sim2.tfm[x] for x in range(len(res_sim2.tft))]
# Flow rate case
res_pred3 = model_lgbm.predict([[t1,q2,rhof1,x] for x in range(0,3050,50)])
res_pred3 = [res_pred3[x] + res_sim3.tfm[x] for x in range(len(res_sim3.tft))]
# Density change
res_pred4 = model_lgbm.predict([[t1,q1,rhof2,x] for x in range(0,3050,50)])
res_pred4 = [res_pred4[x] + res_sim4.tfm[x] for x in range(len(res_sim4.tft))]

# Using the cases below we can create a correction profile, just check the difference in values for each case between
# predicted and simulated, and then calculate the coefficients for the correction function:
dif1 = [res_pred1[x]-res_sim1.tft[x] for x in range(len(res_sim1.tft))]
dif2 = [res_pred2[x]-res_sim2.tft[x] for x in range(len(res_sim2.tft))]
dif3 = [res_pred3[x]-res_sim3.tft[x] for x in range(len(res_sim3.tft))]
dif4 = [res_pred4[x]-res_sim4.tft[x] for x in range(len(res_sim4.tft))]

def get_correction(t, q, rhof):
    m_time = [((dif2[x] - dif1[x]) / (t2 - t1)) for x in range(len(md))]
    m_q = [((dif3[x] - dif1[x]) / (q2 - q1)) for x in range(len(md))]
    m_rhof = [((dif4[x] - dif1[x]) / (rhof2 - rhof1)) for x in range(len(md))]

    correction = [dif1[x] + m_time[x] * (t - t1) + m_q[x] * (q - q1) + m_rhof[x] * (rhof - rhof1) for x in
              range(len(dif1))]

    return correction

# Results
# Now that we have already our correction function, let's use it with the prediction model.
# Set new parameters for circulation time, flow rate and density:
t = 16
q = 260
rhof = 1.24

# Generate simulation a prediction to do the comparison. Additionally we can check the execution time involved for each
# one.

# Simulation
%%time
res_sim5 = pti.temp(t, change_input={'q':q, 'rhof':rhof, 'ts':20, 'tin':25})

# Prediction
%%time
```

```
res_pred5 = model_lgbm.predict([[t,q,rhof,x] for x in range(0,3050,50)])
res_pred5 = [res_pred5[x] + res_sim5.tfm[x] for x in range(len(md))]
correction = get_correction(t, q, rhof)
res_pred5_corrected = [res_pred5[x] - correction[x] for x in range(len(correction))]

# Plotting Simulated vs Predicted
name = [res_sim5, res_pred5, res_pred5_corrected]
md = list(range(0,3050,50))
r2_2 = r2_score(res_sim5.tft, res_pred5_corrected)
plt.plot(name[0].tft, md, c='r', label='Simulated') # Temp. inside Tubing vs Depth
plt.plot(name[2], md, c='b', label='Predicted, R2: {:.4f}'.format(r2_2)) # Temp. inside Tubing vs Depth
plt.xlabel('Temperature, °C')
plt.ylabel('Depth, m')
plt.ylim(0, md[-1])     # bottom and top limits
plt.ylim(plt.ylim()[::-1])  # reversing y axis
plt.legend()  # applying the legend
plt.grid()
plt.show()
```

Appendix F Formation Temperature – Prediction Model

F.1. Assuming no difference in position of WH and Target

```
import petrodc.npd as dc
import numpy as np
import seaborn as sns
import pandas as pd
import os
import json
import itertools
from sklearn.metrics import mean_squared_error, r2_score #for checking the model accuracy
from sklearn.model_selection import cross_val_score, train_test_split
import lightgbm as lgb
%matplotlib inline
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
from matplotlib import cm
import plotly.graph_objects as go
from math import ceil, floor
from statistics import mean
import joblib



# Data acquisition

# Importing data for exploration wells as a dataframe by using petrodc:

df_exp = dc.wellbore(12)

# Let's use only the relevant columns.

df_exp =
df_exp[['wlbWellboreName','wlbMainArea','wlbWaterDepth','wlbFinalVerticalDepth','wlbTotalDepth','wlbNsUtm'
,
              'wlbEwUtm', 'wlbBottomHoleTemperature']]

df_exp.replace('', np.nan, inplace=True)    # Tagging missing data
print('There is data of ',len(df_exp), 'wells')

# Selecting wells within North Sea
df_exp = df_exp[df_exp.wlbMainArea == 'NORTH SEA']
df_exp.drop(['wlbMainArea'], axis=1, inplace=True)   # Drop the column wlbMainArea
df_exp.drop(['wlbWellboreName'], axis=1, inplace=True)   # Drop wellbore name
df_exp = df_exp.astype(float)
print('But only ',len(df_exp), 'wells are located within the North Sea')
```

```
# Now that we know the model is accurate enough, it is possible to generate the seabed surface for within the sector
# analyzed. Let's create the grid using the location and then predict the values for the water depth.

# Defining the limits of the grid:
Ns_lower_limit = floor(df_exp.wlbNsUtm.min()/1000)
Ns_upper_limit = ceil(df_exp.wlbNsUtm.max()/1000)
Ew_lower_limit = floor(df_exp.wlbEwUtm.min()/1000)
Ew_upper_limit = ceil(df_exp.wlbEwUtm.max()/1000)
print('Ns from ', Ns_lower_limit, ' to ', Ns_upper_limit)
print('Ew from ', Ew_lower_limit, ' to ', Ew_upper_limit)

# Generate the values for Ew and Ns:
east = np.arange(Ew_lower_limit, Ew_upper_limit + 1)
north = np.arange(Ns_lower_limit, Ns_upper_limit + 1)

# Generate the grid with the combination between Ew and Ns:
location = [north, east]
grid = np.array(list(itertools.product(*location))) * 1000  # values should be in meters when using the prediction model

# Plot wells with available data
plt.plot(df_exp['wlbEwUtm'].astype(float) / 1000, df_exp['wlbNsUtm'].astype(float) / 1000, 'go', label='All wells')
plt.xlabel('East-utm, km')
plt.ylabel('North-utm, km')
plt.title('Location of wells')
plt.legend()
plt.xlim(300, 700)
plt.show()

# Filtering (Drop rows with NaN and where Bottom Hole Temperature is 0)
df_exp = df_exp[['wlbNsUtm', 'wlbEwUtm', 'wlbFinalVerticalDepth', 'wlbWaterDepth',
            'wlbBottomHoleTemperature']].astype(float)
df_exp.dropna(inplace=True)
df_exp['wlbBottomHoleTemperature'] = df_exp.wlbBottomHoleTemperature.astype(float)
df_exp.drop( df_exp[ df_exp['wlbBottomHoleTemperature'] == 0 ].index , inplace=True)
df_exp = df_exp.reset_index(drop=True)
print('Working with ',len(df_exp), ' wells at this point')

# The figures below show the proportion between the number of wells being currently used and the initial amount of wells
# 2D Plane
plt.plot(df_exp['wlbEwUtm'] / 1000, df_exp['wlbNsUtm'] / 1000, 'go', label='All wells')
plt.xlabel('East-utm, km')
plt.ylabel('North-utm, km')
plt.title('Location of wells')
plt.legend()
plt.xlim(300, 700)
plt.show()
# 3D view
east_new = df_exp.wlbEwUtm / 1000
north_new = df_exp.wlbNsUtm / 1000
df_exp['wlbWaterDepth'] = df_exp.wlbFinalVerticalDepth.astype(float)
```

```
fig = plt.figure()
ax = mplot3d.Axes3D(fig)
ax.view_init(azim=250, elev=30)
surf = ax.scatter(east_new, north_new, df_exp.wlbFinalVerticalDepth)
ax.set_xlabel('Ew-utm, km')
ax.set_ylabel('Ns-utm, km')
ax.set_zlabel('TVD, m')
ax.set_ylim3d(6200, 6900)
ax.invert_zaxis()
plt.show()

# Selecting Training/Testing Datasets
train, test = train_test_split(df_exp, test_size=0.3, shuffle = True)
train_x = train.drop(['wlbBottomHoleTemperature'], axis=1)
train_y = train.wlbBottomHoleTemperature.values
test_x = test.drop(['wlbBottomHoleTemperature'], axis=1)
test_y = test.wlbBottomHoleTemperature.values
print(len(train_y), ' wells are used for training and ', len(test_y), ' for testing')

# CREATING LGBM MODEL
model_lgbm = lgb.LGBMRegressor(objective='regression', num_leaves=5,
                    learning_rate=0.05, n_estimators=400,
                    max_bin = 55,
                    min_data_in_leaf = 2, min_sum_hessian_in_leaf = 2)
score = cross_val_score(model_lgbm, train.values, train_y, cv = 5)
print('Cross Validation',score)
print("LGBM score: {:.4f} ({:.4f})\n" .format(score.mean(), score.std()))

model_lgbm.fit(train_x, train_y)
prediction_lgbm = model_lgbm.predict(test_x.values)
print('Mean Squared Error: {:.4f}'.format(mean_squared_error(test_y, prediction_lgbm, squared=False)))
r2_lgbm = round(r2_score(test_y, prediction_lgbm), 4)
print('R-squared: {:.4f}'.format(r2_lgbm))

lgbm_label = 'LGBM, R2: {:.2f}'.format(r2_lgbm)
plt.plot(list(range(20)), prediction_lgbm[:20], 'b-', label=lgbm_label)
plt.plot(list(range(20)), test_y[:20], 'go', label='True Data')
plt.xlabel('Data point')
plt.ylabel('Formation Temperature, °C')
plt.title('True Data vs Prediction')
plt.legend()
plt.show()

# Saving the model
joblib.dump(model2_lgbm,'tfm_model_lgbm.pkl')

# Loading the model
tfm_model_lgbm = joblib.load('tfm_model_lgbm.pkl')

# CREATING KNN MODEL
```

```
scaler = MinMaxScaler(feature_range=(0, 1))
train_x_scaled = scaler.fit_transform(train_x)
train_x = pd.DataFrame(train_x_scaled)
test_x_scaled = scaler.fit_transform(test_x)
test_x = pd.DataFrame(test_x_scaled)

r2_val=[0]
k_best = None
model_knn = None
prediction_knn = None
for K in range(1, 31):
    model2_knn = neighbors.KNeighborsRegressor(n_neighbors = K)

    model2_knn.fit(train_x, train_y)  #fit the model
    pred=model2_knn.predict(test_x) #make prediction on test set
    calc_r2 = r2_score(test_y,pred)
    if calc_r2 > max(r2_val):
        k_best = K
        model_knn = model2_knn
        prediction_knn = pred
    r2_val.append(calc_r2)
    print('R2 value for k= ' , K , 'is:', calc_r2)

r2_knn = round(max(r2_val), 4)

print('R^2 with LGBM: ', r2_lgbm)
print('R^2 with KNN: ', r2_knn)

# Saving the model
joblib.dump(model_knn,'tfm_model_knn.pkl')

# Loading the model
tfm_model_knn = joblib.load('tfm_model_knn.pkl')

# Comparison of performances (LGBM and KNN prediction models)
lgbm_label = 'LGBM, R2: {:.2f}'.format(r2_lgbm)
knn_label = 'KNN, R2: {:.2f}'.format(r2_knn)
plt.plot(list(range(20)), prediction_lgbm[:20], 'b-', label=lgbm_label)
plt.plot(list(range(20)), prediction_knn[:20], 'r-', label=knn_label)
plt.plot(list(range(20)), test_y[:20], 'go', label='True Data')
plt.xlabel('Data point')
plt.ylabel('Formation Temperature, °C')
plt.title('True Data vs Prediction')
plt.legend()
plt.show()

# With the lgbm model we can generate a good aproximation of the formation temperature for certain locations within
# the North Sea. Let's predict the formation temperature for the grid used previously, for a vertical depth (tvd)
# of 3000 m.
grid_new = pd.DataFrame(grid).rename(columns={0: 'wlbNsUtm', 1: 'wlbEwUtm'})
```

```
grid_new['tvd'] = [3000] * len(grid_new)
grid_new['water_depth'] = df_exp.wlbWaterDepth
grid_new['fm_temp'] = tfm_model_lgbm.predict(grid_new)

# Plotting the results
east_new = grid_new.wlbEwUtm / 1000
north_new = grid_new.wlbNsUtm / 1000

fig = plt.figure()
ax = mplot3d.Axes3D(fig)
ax.view_init(azim=240, elev=40)
surf = ax.plot_trisurf(east_new, north_new, grid_new.fm_temp, cmap=cm.jet, linewidth=0.1)
fig.colorbar(surf, shrink=0.5, aspect=5)
ax.set_xlabel('Ew-utm, km')
ax.set_ylabel('Ns-utm, km')
ax.set_zlabel('Formation Temperature, °C')
ax.set_title('Formation Temperature at tvd = 3000 m')
ax.invert_zaxis()
plt.show()

# The plot above shows that we can get very different formation temperatures for the same vertical depth but different
# locations within the same sector, North Sea in this case.
fig = go.Figure(data =
   go.Contour(
      colorscale='Hot',
      line_smoothing=0.85,
      z=grid_new.fm_temp,
      x=east_new, # horizontal axis
      y=north_new # vertical axis
   ))
fig.update_layout(
   title={
      'text': "Formation Temperature at tvd = 3000 m",
      'y':0.9,
      'x':0.5,
      'xanchor': 'center',
      'yanchor': 'top'},
   xaxis_title="Ew-utm, km",
   yaxis_title="Ns-utm, km",
   width=500,
   height=500)
fig.show()
```

F.2. Loading well bottom locations

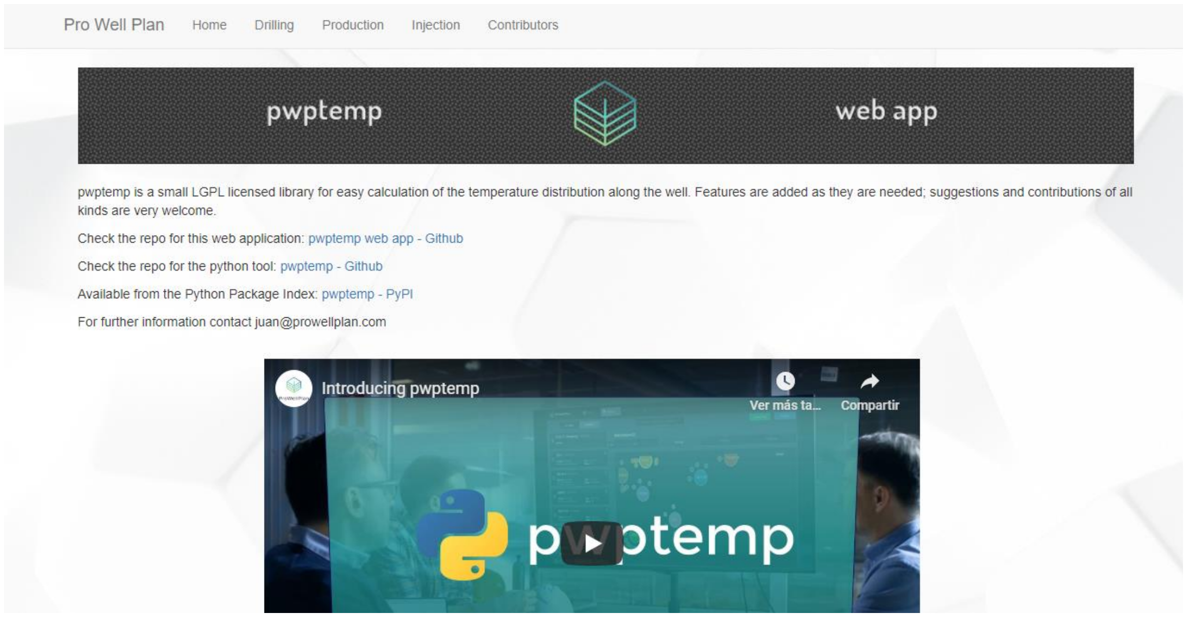
```
# Let's create a new dataset with the wells of which the bottom location is available (let's name it as
# df_exp_bottomloc).

# Change the format of the wellbore names in order to fit the .json files
df_exp['wlbWellboreName'] = [w.replace('/', '_') for w in df_exp.wlbWellboreName]
df_exp['wlbWellboreName'] = [w.replace(' ', '_') for w in df_exp.wlbWellboreName]

# Creating a new column with a boolean for existing wellpath data, True if exist and False if not
df_exp['WellpathData'] = np.array([os.path.exists(name+'.json') for name in df_exp['wlbWellboreName']])

# Take the wells that have wellpath data
df_exp_bottomloc = df_exp[df_exp['WellpathData'] == True].reset_index(drop=True)

print('We have ', len(df_exp_bottomloc), ' wells of ', len(df_exp), ' with bottom location data available.')

# Let's import the bottom location data and add it to the dataset
Ns_bottom = []
Ew_bottom = []
n = []
e = []
for name in df_exp_bottomloc.wlbWellboreName:
    ni = 0
    ei = 0
    with open(name + '.json') as json_file:
        data = json.load(json_file)
        data = json.loads(data[0]['data'])[-1]

    if 'northing' in data.keys():
        Ns_bottom.append(data['northing'])
        ni = 1
    else:
        if 'norhting' in data.keys():
            Ns_bottom.append(data['norhting'])
            ni = 1
        if 'utm n' in data.keys():
            Ns_bottom.append(data['utm n'])
            ni = 1
    if ni == 0:
        if 'north_offset' in data.keys():
            Ns_bottom.append(
                float(df_exp_bottomloc.wlbNsUtm[df_exp_bottomloc.wlbWellboreName == name]) + data['north_offset'])
            ni = 1
        else:
            if 'ndist' in data.keys():
                Ns_bottom.append(
                    float(df_exp_bottomloc.wlbNsUtm[df_exp_bottomloc.wlbWellboreName == name]) + data['ndist'])
                ni = 1
            if 'nort' in data.keys():
                Ns_bottom.append(
                    float(df_exp_bottomloc.wlbNsUtm[df_exp_bottomloc.wlbWellboreName == name]) + data['nort'])
                ni = 1

    if 'easting' in data.keys():
        Ew_bottom.append(data['easting'])
        ei = 1
    else:
        if 'east_offset' in data.keys():
            Ew_bottom.append(
                float(df_exp_bottomloc.wlbEwUtm[df_exp_bottomloc.wlbWellboreName == name]) + data['east_offset'])
            ei = 1
        if 'edist' in data.keys():
            Ew_bottom.append(float(df_exp_bottomloc.wlbEwUtm[df_exp_bottomloc.wlbWellboreName == name]) +
data['edist'])
            ei = 1

    n.append(ni)
    e.append(ei)

    if ni == 0:
        Ns_bottom.append(np.nan)  # Add NaN if there is not any value for northing
    if ei == 0:
        Ew_bottom.append(np.nan)  # Add NaN if there is not any value for easting

# Adding the bottom hole location to the dataset
df_exp_bottomloc = df_exp_bottomloc.join(pd.DataFrame({'Ns_bottom': Ns_bottom,'Ew_bottom': Ew_bottom}))
print('checking the amount of missing data NaN:')
print(df_exp_bottomloc.isnull().sum())

# Now let's drop the wells with NaN for the location data.
df_exp_bottomloc.dropna(inplace=True)

# Some data in Ns_bottom is negative, let's drop it
df_exp_bottomloc.drop( df_exp_bottomloc[ df_exp_bottomloc['Ns_bottom'] < 0 ].index , inplace=True)
```

Appendix G Web-based Application

https://github.com/pro-well-plan/pwptemp-web-app

G.1. Home page

http://pwptemp.appspot.com/



pwptemp is a small LGPL licensed library for easy calculation of the temperature distribution along the well. Features are added as they are needed; suggestions and contributions of all kinds are very welcome.

Check the repo for this web application: pwptemp web app - Github

Check the repo for the python tool: pwptemp - Github

Available from the Python Package Index: pwptemp - PyPI

For further information contact juan@prowellplan.com

G.2. Drilling module

http://pwptemp.appspot.com/drilling

G.3. Production module

http://pwptemp.appspot.com/production

G.4. Injection module

http://pwptemp.appspot.com/injection