

CFD Study of Emergency Evacuation of Molten Salt in a Nuclear Reactor

Viktor Akse^{a,b}, Daniel Ø. Halvorsen^{a,b}, Agsagan Ragunathan^{a,d}, Vegard S. Skjefstad^{a,c}

^aNorwegian University of Science and Technology, 7491 Trondheim, Norway

^bDepartment of Physics, Høgskoleringen 5

^cDepartment of Structural Engineering, Richard Birkelands v 1A

^dDepartment of Energy and Process Engineering, Kolbjørn Hejes v 1B

Abstract

This study looked at the time transient and temperature variations of a molten salt mixture as it is passively drained from the reactor core of a hypothetical molten salt nuclear reactor (MSR). The computational fluid dynamics (CFD) software OpenFOAM, was used to study the temperature development with a simplified cylinder geometry. The RANS model $k - \omega SST$ was used together with 2D axisymmetry and volume of fluid (VOF) multiphase. According to our simulations, the temperature increases from 1023 K to 1068 K within 164.5 s where 93.2 % of the molten salt was drained. The critical temperature threshold of 1073 K was not reached.

1. Introduction

Nuclear power refers to the use of nuclear reactions to release nuclear energy in the form of heat. In most nuclear power plants today, this heat is used to convert water to steam, which in turn drives a turbine to produce electricity. This was first realised in 1948, when a nuclear power station in Oak Ridge, Tennessee, was used to power a light bulb [1]. Ever since, nuclear power stations have been used extensively around the world to power our growing energy demands. Today, nuclear power is responsible for the creation of about 10 % of global electricity use, and only lags hydro-power as a low-emission power source [2].

Historically, one of the main reasons for the widespread commercialization of nuclear power as an energy source, is due to its high utility of operation when compared to other energy sources. The net capacity factor represents the ratio of real electrical energy output over a time period to the maximum possible such output over the same period [3], and for nuclear reactors this factor averages around 91 %. This is significantly higher than for other energy forms like solar and wind, which in 2018 reported net capacity factors of 25.1 % and 34.6 % respectively [4, 5]. Nuclear power is also a reliable, and contrary to popular belief, safe option when compared to other traditional energy sources. Including all known nuclear accidents, the average amount of deaths linked to power production lies at 0.07 deaths per TWh, whilst for oil it lies at 18.43 [4]. Nuclear power has another significant advantage to it, which is of great interest in our time; its minuscule carbon footprint. It is estimated that since its global adaption in the 1970's, nuclear power has prevented the emission of around 64 gigatonnes of CO_2 equivalent greenhouse gasses [6]. France is a country which currently derives about 75 % of the power it consumes domestically from nuclear energy, and as a result enjoys low levels of CO_2 emissions [7]. In 2014

for instance, it produced about 4.5 tons of CO_2 emissions per capita, almost half that of 8.9 tons produced by Germany in the same year [8]. Nuclear power however, does not come without its disadvantages. The two most significant issues are to do with nuclear waste and nuclear proliferation. Currently, all nuclear reactors produce a by-product called radioactive waste. This refers to the array of substances that nuclear fuel transmutes into after it has passed through a reactor, the most dangerous of which have the potential to cause great harm to biological matter due to their radioactivity. Though only about 3 % of total radioactive waste coming from a reactor is considered extremely dangerous, it can remain radioactive for hundreds of thousands of years [9, 10]. There are plans for large subterranean storage facilities for storing such waste, one of which is currently under construction in Finland [11]. However, it is practically impossible to plan for a scenario hundreds of thousands of years in the future, so by this logic, waste-storage is a temporary solution at best.

The problem of nuclear proliferation stems from Plutonium, which is a common by-product produced by most reactors operating today as a part of their nuclear waste products [12]. Plutonium, if separated from the waste, can be used as an ingredient in the manufacture of nuclear warheads [12]. Most new generation 4 reactor designs currently being researched are able to largely get around this problem by a variety of methods, like making sure plutonium is never separated from other waste materials in the first place [13].

One type of nuclear reactor that has seen a recent renaissance in R&D, is the molten salt reactor (MSR), first tested at the Oak Ridge National Laboratory in 1965 [14]. Operating at atmospheric pressure, it uses a molten salt mixture as both its fuel and coolant, leading to greater efficiency and a higher safety standard [15]. The Danish

startup company Seaborg Technologies is currently designing its own compact version of a MSR, this reactor will be able to reprocess radioactive waste as part of its fuel cycle [16]. As with other MSR's, Seaborg's version will also need a system to ensure that the radioactive fuel salt could drain passively in the case of an emergency such as a total electrical failure. The aim of this project is to investigate the process of such a passive draining of fuel salt from the reactor core, and to evaluate the time transient and temperature variations of the salt mixture during this process. This will be done with the aid of the open-source CFD tool OpenFOAM, employing the Navier-Stokes, mass, energy and Wigner-Way equations.

2. Method

The MSR in focus is based on the proposed design by Seaborg Technologies. The reactor consists of a core tank where the splitting of atoms, known as fission, happens in the fuel and heat is produced. It also contains a set of heat exchange circuits, where thermal energy is transferred from the core to a water reservoir, which subsequently forms steam and drives a turbine. In this project, the internal geometry of the core has been simplified to a cylinder with a drainage pipe, and only this will be considered in the calculations to follow. Below the core is the drainage tank where the fuel will be drained to in the event of an emergency. The drainage tank have been omitted in our simulation. The drainage pipe separates the core and drainage tank, and inside is an actively cooled freeze-plug which will quickly melt in the event of power failure. The drainage of the fuel from the core into the drainage tank will then be passively powered by gravitation alone. In our simulations, the presence of a freeze-plug has been ignored. Subsequently, the pipe is initially filled with the fuel. To ensure drainage gas will fill the void left behind by the fuel.

During evacuation the fuel is still generating heat and will no longer be actively cooled. It is therefore paramount that the fuel is drained before the temperature rises above the temperature threshold of 1073 K [17], which is when the piping material will begin to degrade. The heat generation can be viewed as homogeneous in the fuel and the drainage time is therefore the critical parameter which determines the temperature development in the core assuming an adiabatic system.

To evaluate the performance of the evacuation procedure we will therefore determine i) drainage time and ii) temperature development during drainage.

2.1. Model approach

As this is a part of the initial numerical study of the drainage time and temperature development in the system, the focus will be on the flow inside the simplified core geometry and outlet pipe. The computational domain is thus a cylinder of diameter 2 m with a vertical drainage

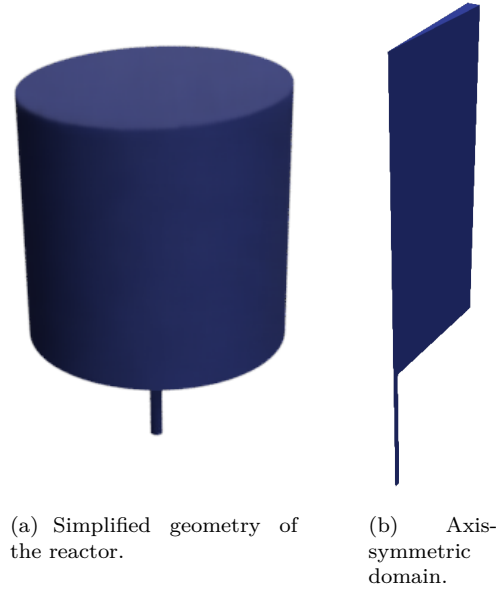


Figure 1: Reactor geometry and axis-symmetric domain

pipe below of diameter 0.1 m and height 1 m. Initially the core is assumed completely filled with fuel, resulting in the liquid surface lying 3 m above ground level. The tank height was extended by 0.3 m to ensure that the fuel does not spill out to the open atmosphere at the top. The drainage pipe is placed in the center of the bottom surface of the core. The geometry can be viewed in 1(a).

The freeze plug is located at the top of the drainage pipe meaning at initial drainage the pipe is not filled. However, our simulation was initiated with the pipe filled with fuel in order to more accurately validate our results to V. M. Akse [17] and for the simulation to run more smoothly.

Due to the cylindrical geometry of the computational domain we have chosen to use an axisymmetric solver. This will severely reduce the computational time of the simulations in contrast to a full 3D mesh. Using axisymmetry means that details in θ -direction will be neglected. As the tank is passively drained by gravitation in z -direction, orthogonal to the θ -direction, the only detail that is expected to be lost is the swirling motion which is a common phenomenon in drainage. The domain is constructed according to an OpenFOAM guide [18] with a wedge-angle of 5° shown in figure 1(b).

The model constructed in OpenFOAM consists of two steps; i) solving for drainage time based on constant fluid properties, ii) solving for drainage time based on temperature dependent fluid properties and including the energy equation to simulate the temperature development. In both cases we are using multiphase VOF simulations. Multiphase is used as the behaviour of fluid being drained does not require adaptive mesh. VOF is used as gas does not mix with fuel and is therefore immiscible. In the first step, *interFoam* solver is used, and for the second *compressibleInterFoam*. Different solvers are implemented

as *interFoam* is easy to work with and can quickly obtain good answers, but it lacks the energy equation and cannot work with a change in the properties of the medium.

Turbulent flow will occur in the drainage pipe, whilst it will remain laminar in the core. From simple Bernoulli equation (1) results with $Re = 3.109 \cdot 10^5$ at the outlet pipe. With this mixture of laminar and turbulent flow, $k-\omega$ SST RANS model is used as it is well suitable to capture the transition from laminar to turbulent flow behaviour [19]. Eddies are not expected to impact the flow behaviour in a large degree which is why we have chosen a RANS model over e.g. LES or DNS models to save computational time.

$$\frac{1}{2}u^2 + \frac{p}{\rho} + gz = \text{constant}. \quad (1)$$

2.2. $k-\omega$ Shear Stress Transport turbulence model

The turbulence model used in this project is a two-equation eddy-viscosity model which combines the $k-\omega$ model and the $k-\epsilon$ model. Its purpose is to capture the physics of both the inner region of the boundary layer using the $k-\omega$ model and the free shear flow using the $k-\epsilon$ model. The combination of the two models ensures that effects of adverse pressure gradients and separated flow are captured [20, 21]. The two transport equations for the turbulent kinetic energy, k and the rate of dissipation of eddies, ω , are given by

$$\begin{aligned} \frac{D}{Dt}(\rho\omega) = & \nabla \cdot (\rho D_\omega \nabla \omega) + \frac{\rho\gamma G}{\nu} - \frac{2}{3}\rho\gamma\omega(\nabla \cdot \mathbf{u}) \\ & - \rho\beta\omega^2 - \rho(F_1 - 1)CD_{k\omega} + S_\omega, \end{aligned} \quad (2)$$

and

$$\frac{D}{Dt}(\rho k) = \nabla \cdot (\rho D_k \nabla k) + \rho G - \frac{2}{3}\rho k(\nabla \cdot \mathbf{u}) - \rho\beta^* \omega k + S_k. \quad (3)$$

ρ is the density, \mathbf{u} is the velocity field, CD_ω , F_1 , D_ω , S_ω and S_k are closure coefficients and auxiliary relations. The model coefficients of equation (2) and (3) are listed in table 1. For initialization the turbulent kinetic energy, k is given by

$$k = \frac{3}{2}(UI)^2, \quad (4)$$

where I is the turbulence intensity which is defined as

$$I = \frac{u'}{U}, \quad (5)$$

where u' is the root-mean-square of the turbulent velocity fluctuations and U is the mean velocity in the flow. The turbulent dissipation rate is given by

$$\omega = C_\mu^{-\frac{1}{4}} \frac{\sqrt{k}}{l}, \quad (6)$$

where C_μ is a turbulence model constant which often takes the value 0.09, k is the turbulent kinetic energy as defined

in (4) and l is the turbulent length scale which describes the large energy containing eddies in the turbulent flow. In fully developed pipe flows, l is related to the pipe diameter by $l = 0.07d$ [22].

2.3. Wall functions

For turbulent flows the boundary layer created by the no-slip condition at the wall is more complicated than in laminar flows. The stresses imposed on the fluid in the boundary layer is contributed to by both viscous and turbulent stresses. The boundary layer is divided in three parts; the viscous sub-layer where viscous stresses are dominant, the log-law region where turbulent stresses are dominant, and the buffer region where there is a mix of viscous and turbulent stresses. For near wall treatment the following non-dimensional variables u^+ and y^+ are defined;

$$y^+ = \frac{yu_\tau}{\nu}, \quad u^+ = \frac{u}{u_\tau}, \quad u_\tau = \sqrt{\frac{\tau_w}{\rho}}$$

where y is the distance from the wall, ν the kinematic viscosity of the fluid, u the velocity parallel to the wall, τ_w the shear stress at the wall.

How u^+ varies with y^+ is shown in figure 2. In the viscous sublayer the relation is linear, while in the log-law region the relation is logarithmic. In the buffer region neither function is correct, and an approximation between the two are used [23].

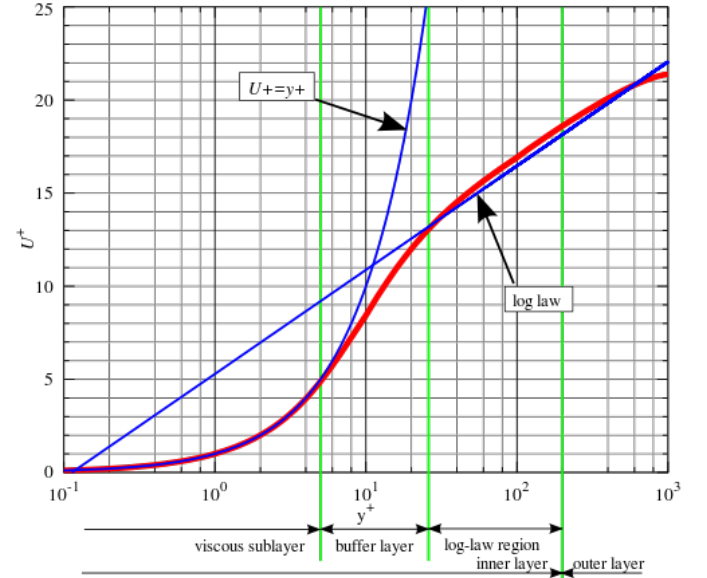


Figure 2: u^+ varying y^+ . Blue lines are the linear and logarithmic relations. Red line is the actual relationship. Areas in the boundary layer based on y^+ is shown. Figure taken from [24].

Based on y^+ values, one can determine where in the boundary layer the mesh cell at the wall in your simulation is and what kind of information is lost which is approximated by wall functions.

Table 1: Model coefficients used in the $k - \omega$ SST model.

Variables	α_{k1}	α_{k2}	$\alpha_{\omega1}$	$\alpha_{\omega2}$	β_1	β_2	γ_1	γ_2	β^*	α_1	β_1	c_1
Value	0.85	1.0	0.5	0.856	0.075	0.0828	5/9	0.44	0.09	0.31	1.0	10.0

There are three different wall functions implemented in this simulation, corresponding to three different variables. For the turbulent kinetic energy, k , the *kqRWallFunction* is used, and it imposes a zero gradient condition on the boundary. The turbulent dissipation rate, ω , uses the function *OmegaWallFunction*, which combines the viscous equation and logarithmic equation. It switches between the regions according to the y^+ values. It is valid in the region $y^+ \in [30, 300]$. The last wall function used is for the eddy viscosity, ν_t , which uses *nutkWallFunction*. This function determines the eddy viscosity by the following relations

$$\begin{aligned} y^+ > y_{lam}^+ : \quad \nu_t &= \nu \left(\frac{ky^+}{\ln(Ey^+)} - 1 \right), \\ y^+ < y_{lam}^+ : \quad \nu_t &= 0, \end{aligned} \quad (7)$$

where the variable y_{lam}^+ determines the transition from viscous layer to logarithmic layer. Its value is approximated to 11.5. E is a roughness coefficient equal to 9 [23].

2.4. Boundary conditions

The walls of the reactor are treated with simple no-slip and adiabatic wall conditions for *compressibleInterFoam*. Adiabatic walls are chosen as we have no data of the wall material and thickness and the ambient temperature. The heat transported through the wall is also considered to be small compared to the heat generation in the fuel. The wedge walls are treated with axisymmetry condition where $\frac{\partial X}{\partial \theta} = 0$, X being any variable.

At the inlet, where gas enters the core, and at the outlet, set as the bottom of the drainage pipe, the pressure is set at atmospheric pressure. The velocity at the inlet and outlet is set to a pressure based inlet-outlet condition.

2.5. Initial conditions and physical properties

The physical values of the fuel and filler gas are given in Table 2. The properties of the filler gas are assumed to be identical to air, as we have no data for it. The initial temperature in the reactor is set at 1023 K for both fuel and gas, which is the temperature that is assumed to be reached by the time passive drainage commences.

Both dynamic viscosity and density of fuel and gas are temperature dependent. The change of fuel properties are given by the following equations which are provided by Seaborg.

$$\mu_f = 7.7 \cdot 10^{-5} e^{\frac{4444.444}{T}} \quad (8)$$

$$\rho_f = 3606.4 - 0.201T \quad (9)$$

Where μ_f and ρ_f are the dynamic viscosity and density of the fuel respectively, and T the temperature in [K]. The change of viscosity of the gas is assumed to be negligible and set to constant value at 1023 K while the density is determined by ideal gas law of air

$$\rho_g = \frac{P}{RT} \quad (10)$$

where ρ_g is the density of gas, P is the pressure, R is the specific gas constant for air.

Table 2: Physical values of gas and fuel

Medium	$\rho[\frac{Kg}{m^3}]$	$\mu[Pa \cdot s]$	$cp[\frac{J}{Kg \cdot K}]$	$k[\frac{W}{m \cdot K}]$
Fuel	(9)	(8)	1360	1.19
Gas	(10)	$4.21 \cdot 10^{-5}$	1143	1.33

There is expected to be initial velocity of the fuel due to the working pumps. However as we have no way of determining the initial velocity it has been assumed to be zero. These velocities would be partially directed in θ -direction meaning that the axisymmetry model would not be feasible.

2.6. Heat production

The heat produced by the fuel is due to radioactive decay given by the Wigner-Way formula [25–27]

$$\frac{P}{P0} = 5 \cdot 10^{-3} a [t_{elapsd}^{-b} - (t_s + t_{elapsd})^{-b}], \quad (11)$$

where P is the total heat produced in watts, $P0$ is the initial heat production immediately prior to shutdown in watts, t_{elapsd} is the time since reactor shutdown and t_s is the time the reactor has been operational. a and b are coefficients that vary by time after shutdown according to the values given in Table 3. Seaborg has provided values for $P0$ and t_s which are 250MW and $8.64 \cdot 10^6 s$ respectively. The heat production given by equation (11) and is presented in figure 3.

Table 3: a and b varying with t_{elapsd} in equation (11).

$t_{elapsd}[s]$	a	b
0.1 - 10	12.05	0.0639
10 - 150	15.31	0.1807
150 - $8 \cdot 10^8$	27.43	0.2962

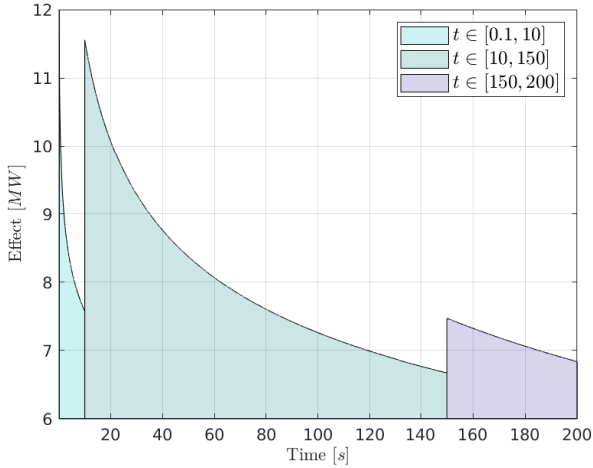


Figure 3: Power output as a function of time according to equation (11) using coefficients from table 3. The model suggests that the heat production will increase discontinuously at $t=10$ s and $t=150$ s. The curve is divided into three color regions indicating the different coefficients.

The heat production is implemented in OpenFOAM as a source term in the heat equation (12) with units $[\frac{Kg \cdot K}{s \cdot m^3}]$. This is achieved by multiplying the power P with $\frac{1}{cpV}$ where cp is the specific heat capacity of the fuel and V the initial volume of fuel in the reactor at shutdown. The source term is applied to the cells in the simulation by multiplying updated values of the fuel fraction in each cell at each time step.

$$\frac{\partial(\rho T)}{\partial t} + \mathbf{u} \cdot \nabla(\rho T) = \frac{k}{cp} \nabla^2 T + \frac{\mu}{cp} \Phi_u + S. \quad (12)$$

Where k the constant thermal conductivity, cp the constant specific heat coefficient, μ the dynamic viscosity, Φ_u viscous dissipation and S the source term.

2.7. Numerical schemes

In OpenFOAM it is possible to choose how the different terms in the transport equations will be approximated using different numerical schemes. The transient terms is solved using an implicit Euler scheme, which is first order accurate and bounded. The time step is determined by the Courant number. The simulations was executed by using three different Courant number threshold values for verification purposes. Gradient terms for velocity and temperature are solved using a cellLimited Gauss linear method which improves stability and boundedness compared to the default scheme Gauss linear, and it's second order accurate. The laplacian terms are solved using the Gauss linear corrected scheme which is second order accurate and conservative, but unbounded. This scheme is more suitable for a non-structured mesh, however, its accuracy is more beneficial for us than the reduced runtime related to the correction of non-orthogonality in the mesh. The divergence terms use either a Gauss linear, Gauss vanLeer or a Gauss limitedLinear scheme. These are all second

order accurate but the vanLeer and limitedLinear includes a limitation in regions of rapidly changing gradient.

2.8. Mesh

The meshing of the domain is structured where grading towards the outlet pipe wall is used. Four mesh versions for verification purposes are created where the cell count is doubled in each direction throughout the domain for each step from rough to fine. The mesh and initial domain of gas and fuel can be viewed in figure 4.

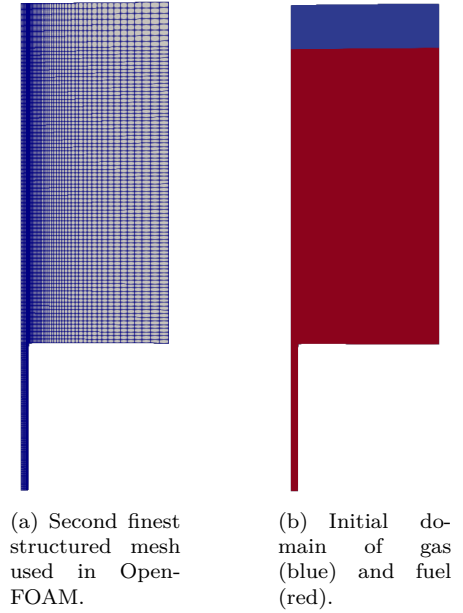


Figure 4: Mesh and initial values of gas and fuel.

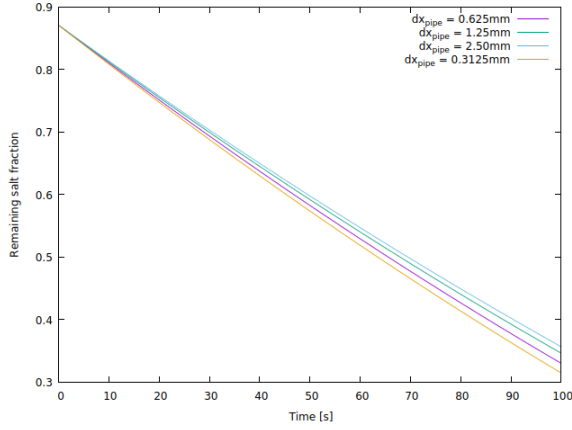
3. Results and Discussion

The results from the *compressibleInterFoam* simulations are presented along with a discussion. Outlet volumetric discharge rate is used to check mesh- and time-convergence and y^+ values are checked to see whether the wall functions used are within the intended working range. V. M. Akse [17] and [28] are used to validate our results.

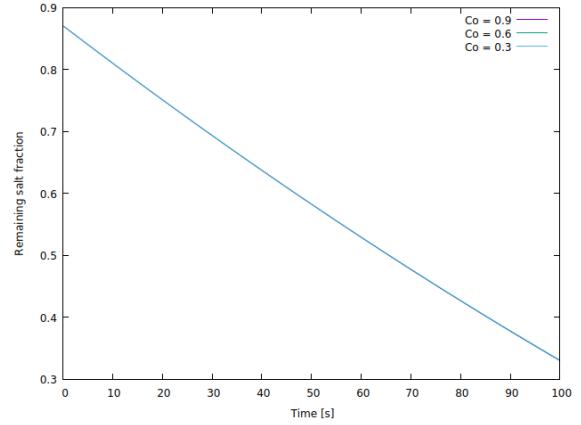
3.1. Mesh- and time-convergence

Four different mesh sizes are used to see how our solution deviates. The criteria used for mesh-convergence is volumetric discharge rate as at a certain fuel height left in the system the solution loses accuracy. In table 4 one can see that the error decreases with finer mesh of order 10^{-2} , however a worrying observation is that the gradient also increases with finer mesh. This indicates that the solution may be far from having mesh-convergence, however, this can be due to the use of wall functions, which will never be able to resolved the boundary layer accurately.

Different thresholds of the Courant number have been run to check the convergence of time steps, which can be



(a) dx_{pipe} denotes the length of mesh cells in radial direction equivalent to mesh size in table 4



(b) Time step determined by Courant number

Figure 5: Fraction of fuel left in the reactor over first 100s compared between mesh (a) and time steps (b)

viewed in table 5. The error decreases with an order of 10^{-3} and has not yet converged.

Table 4: Absolute relative error of the volumetric discharge rate for different mesh sizes. The base comparison is the finest mesh size which is represented with 0 error. Mesh size corresponds to number of cells along radial direction of outlet pipe.

Cell nr	2	4	8	16
Error	0.07483	0.05659	0.02759	0

Table 5: Absolute relative error of the volumetric discharge rate for different Courant number thresholds. The base comparison is the smallest threshold at 0.3 which is represented with 0 error.

Co nr	1.8	0.9	0.6	0.3
Error	0.00264	0.00168	0.00123	0

Figure 5 shows the difference of remaining fuel fraction left in the reactor between mesh versions and time steps used in the simulations. It is evident that the solution is far more sensitive to the mesh size compared to the time step. From this figure one can see that the drainage time decreases with finer mesh and thereby the highest temperature should be less with the finer meshes.

3.2. y^+ and wall function validity

In our simulation we have the *OmegaWallFunction* which is valid in the interval $y^+ \in [30, 300]$ while the other wall functions are valid for any y^+ . The areas where the wall function influences the solution are where there are high velocities in the simulation. These areas are in the outlet

pipe and a small area at the bottom wall of the core close to the pipe. The y^+ -values are shown in table 6. We can see that the finest mesh is most coherent with the wall function validity, however the computational run time increased by about 10 times compare to the second finest mesh. It was therefore decided to use the second finest mesh for our final simulation, however, this error is considered in the discussion and further work. Reason for the high y^+ -values in the core cylinder wall comes from the gas which is of low importance in our simulations.

3.3. Residuals

Residuals in the simulation are shown in figure 6. The solver is set to iterate on the same time step until all residuals are $\leq 10^{-8}$. Common for all residuals is the fact that that from $t=164.5$ s they are more unstable.

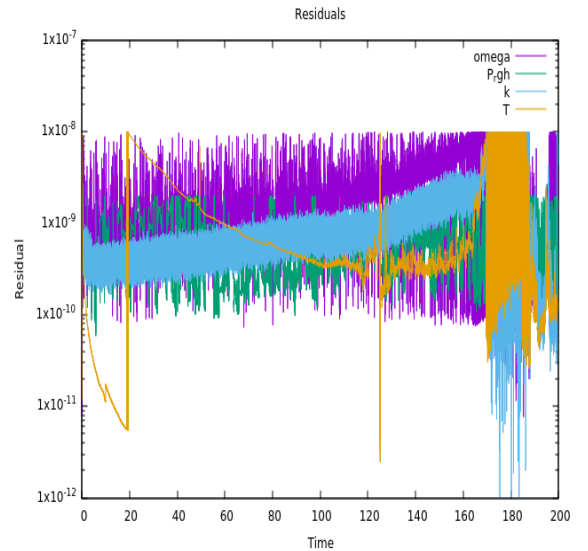
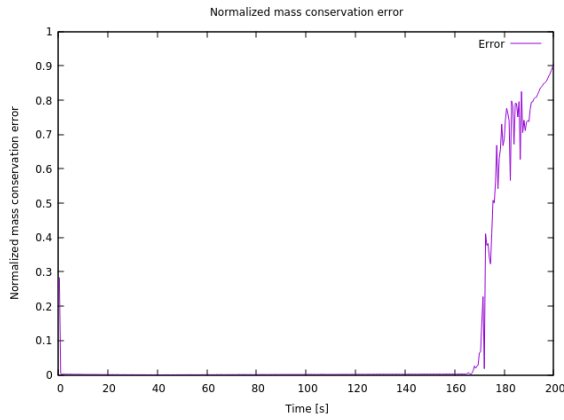


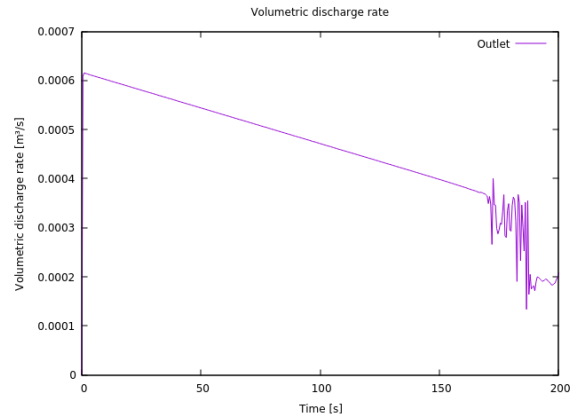
Figure 6: Residuals for the simulation.

Table 6: Range of min, max and average y^+ -values over the first 100 s of simulation for different wall areas and mesh.

Cell nr	Core cylinder wall			Core bottom wall			Outlet pipe		
	y_{min}^+	y_{avg}^+	y_{max}^+	y_{min}^+	y_{avg}^+	y_{max}^+	y_{min}^+	y_{avg}^+	y_{max}^+
2	[4.9, 178]	[101, 5190]	[237, 5750]	[41, 432]	[646, 1877]	[2548, 3341]	[204, 255]	[537, 569]	[391, 660]
4	[1.0, 72]	[37, 2182]	[84, 2413]	[18, 261]	[226, 623]	[907, 923]	[22, 26]	[270, 301]	[330, 371]
8	[0.16, 25]	[12, 802]	[33, 884]	[4.8, 75]	[95, 256]	[536, 621]	[84, 96]	[164, 187]	[193, 218]
16	[0.0037, 6.8]	[4.9, 271]	[13, 299]	[0.24, 31]	[46, 109]	[395, 396]	[40, 48]	[97, 111]	[100, 115]



(a) Mass conservation



(b) Volumetric discharge

Figure 7: Mass conservation and volumetric discharge throughout the simulation. Both plots have instability after $t = 164.5s$

3.4. Solution accuracy

Volumetric discharge and mass conservation is stable until a certain point in the simulation where the fuel left in the reactor is so small that the outlet pipe is no longer fully filled. We have therefore set a cutoff in the results at $t=164.5s$ where this instability occurs and not evaluating the results afterwards. The instability can be viewed in figure 7 where the simulation has been run to $t=200s$ and the fuel left in the reactor at $t=164.5s$ in figure 8. This is also the cause for the instability of the residuals from $t=164.5s$.

From the simulation after $t=164.5s$ one can see that the gas starts to enter the domain from the outlet which is believed to be the reason for the instability in the simulation. The fact that gas enters the outlet when the drainage is near complete was also observed in a simple experiment where water was drained from a small tank through a thin pipe at the bottom.

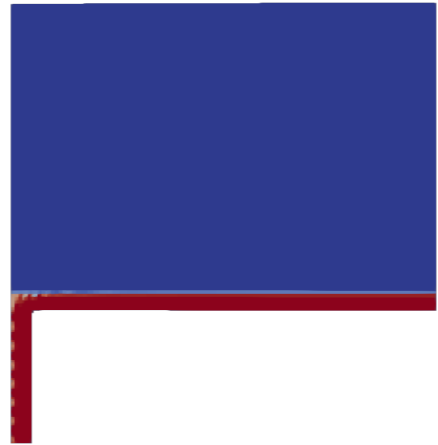


Figure 8: Fuel left in the reactor at $t=164.5s$. Forming of gas in the outlet pipe is barely visible

3.5. Temperature distribution

The temperature distribution as a function of time is presented in figure 9. It is evident that the temperature is increasing due to the positive source term in equation (12) and the gradient gradually decreases as a result from

the decreasing value of the source term. From figure 3 it is clear that the source term is discontinuous at the times $t=10\text{s}$ and $t=150\text{s}$, and the temperature distribution is expected to inherit this effect. This effect is seen as a change of slope at those respective time levels. This is not a physical effect but more a result of the empirical modelling of the source term.

As the simulation is stopped at $t=164.5\text{s}$ due to the increased inaccuracy of mass conservation and volumetric discharge, the maximum temperature in the system is measured at this time step, and its value is 1067.8 K , which is lower than the critical value of 1073 K . This is in agreement with the analytical result presented in V. M. Akse [17]. The tank is however not emptied at this time due to poor ducting.

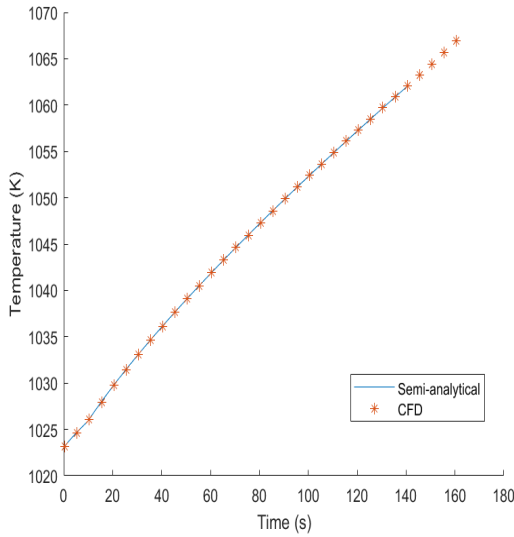


Figure 9: Temperature distribution as a function of time from both the OpenFOAM simulation and the semi-analytical result from V. M. Akse [17].

Another important result seen in figure 9 is that the semi-analytical result from V. M. Akse [17] and the OpenFOAM simulation conducted in this project are near identical in values. The mean two-norm of the two data sets are 0.1155. For this specific CFD simulation with adiabatic wall boundary conditions it would be sufficient to only solve for the drainage time and use a semi-analytical model for the maximum temperature.

The difference for the density and dynamic viscosity for the fuel is determined by equations (9) and (8) respectively. With a ΔT of 44.8 K . The change in density is 0.27% which is negligible, however the viscosity is reduced by 19.98% which would impact the drainage time. This project could have used the *interFoam* solver with an analytic solution of temperature based on iteration time to update the values of the viscosity. The solver would then have one less equation to solve and computational time reduced.

3.6. Validation

For validation we are considering the analytical solution from V. M. Akse [17] with our simulations and how it correlates to similar experiment vs analytical solution in [28]. We estimate the drainage time to reach the same fuel fraction at $t=164.5\text{s}$ with the analytical solution and see how the deviance correlates to C. V. Subbarao [28]. Figure 10 shows the drainage time to reach a fuel fraction of 0.06773.

In our simulations the drainage time takes 28.6% longer than the analytical solution. The fact that the simulation takes longer than the analytical solution fits well with the data from C. V. Subbarao [28] where the analytical solution constantly overpredicts the drainage time compared to the experiment varying between $11\text{--}27\%$ for the cases with closest geometry to our own. C. V. Subbarao [28] does have large geometrical differences than our case and its results can not directly be used to validate our result.

While the analytical solution can take into account the friction loss in the pipe, it does not take into account the losses due to inlet effects from the core to the outlet pipe. Experiments can be made to determine a loss coefficient to be added in the analytical solution, however in order for it to fit our simulation it has to be made with a fluid with similar viscosity and similar tank geometry.

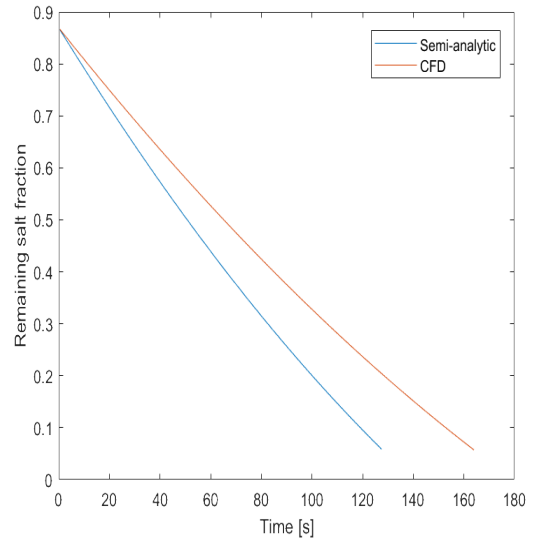


Figure 10: CFD vs analytical drainage time to reach fuel fraction of 0.06773.

4. Conclusion

From our simulations the temperature in the reactor will not reach the critical temperature of 1073 K where the solution is accurate. The solution is considered to be accurate until 164.5s , where 6.7% of the fuel was remaining in the core. Better ducting of the outlet must be made in order for the simulation to be accurate for the entire evacuation procedure and determining the actual drainage time.

The simulation uses 28.6 % longer time to drain than the analytical solution which is agreeable with tank drainage experiments.

The same solution could have been obtained with the *interFoam* solver as the temperature development mimics the analytical solution when using adiabatic walls.

5. Further work

To gain better results in the simulation, further research in mesh convergence is recommended. To better validate the results from the simulation an experiment with fluid of similar properties and similar geometry is recommended.

Adding heat transport through the walls will most likely make the temperature deviate from the analytical solution and should be implemented.

As this model uses axisymmetry it would be interesting to investigate whether swirling motion, which is natural in drainage, will appear in a full 3D simulation. A large cause of error in the simulation is when the fuel is almost completely drained where the gas starts to enter the outlet pipe. A better ducted outlet would delay this effect and possibly eliminate it. Adding a drainage tank on the bottom of the outlet pipe would also be interesting to see whether the change of outlet conditions in the pipe impacts the solution.

Once these uncertainties are determined the next step would be to include a more detailed geometry of the reactor to include the effects from e.g pumps.

Acknowledgments

We would like to thank our lecturer, Reidar Kristofersen, for valuable discussions on implementation of the model and the results. We would also like to thank Seaborg Technologies for providing us with the necessary information and specifications to successfully complete this project.

References

- [1] O. N. Laboratory, Graphite Reactor.
URL <https://www.ornl.gov/content/graphite-reactor>
- [2] IEA.org, Steep decline in nuclear power would threaten energy security and climate goals.
URL <https://www.iea.org/newsroom/news/2019/may/steep-decline-in-nuclear-power-would-threaten-energy-security-and-climate-goals.html>
- [3] U.S.NRC, Capacity factor (net).
URL <https://www.nrc.gov/reading-rm/basic-ref/glossary/capacity-factor-net.html>
- [4] H. Ritchie, It goes completely against what most believe, but out of all major energy sources, nuclear is the safest.
URL <https://ourworldindata.org/what-is-the-safest-form-of-energy>
- [5] E. S. G. International, Know the efficiency of nuclear power.
URL <https://www.esgi.net/2015/05/08/know-efficiency-of-nuclear-power-energy-staffing-jobs/>
- [6] P. A. Kharecha, J. E. Hansen, Prevented mortality and greenhouse gas emissions from historical and projected nuclear power, *Environmental Science Technology* 47 (4889-4895) (2013).
- [7] World-Nuclear.org, Nuclear Power in France.
URL <https://www.world-nuclear.org/information-library/country-profiles/countries-a-f/france.aspx>
- [8] T. W. Bank, CO₂ emissions (metric tons per capita).
URL https://data.worldbank.org/indicator/EN.ATM.CO2E.PC?locations=FR&name_desc=false
- [9] WhatIsNuclear.com, What about the waste?
URL <https://whatisnuclear.com/waste.html>
- [10] world nuclear.org, Radioactive Waste Management.
URL <https://www.world-nuclear.org/information-library/nuclear-fuel-cycle/nuclear-wastes/radioactive-waste-management.aspx>
- [11] A. Kauranen, M. Heinrich, World's first underground nuclear waste storage moves forward in Finland.
URL <https://www.reuters.com/article/us-finland-nuclear-waste/worlds-first-underground-nuclear-waste-storage-moves-forward-in-finland-idUSKCN1TQ1NL>
- [12] P. T. C. Richard L. Wagner Jr., Edward D. Arthur, Plutonium, nuclear power, and nuclear weapons, *ISSUES in Science and Technology* 15 (3) (1999).
- [13] world nuclear.org, Generation IV Nuclear Reactors.
URL <https://world-nuclear.org/information-library/nuclear-fuel-cycle/nuclear-power-reactors/generation-iv-nuclear-reactors.aspx>
- [14] samofar.eu, History of MSR.
URL <http://samofar.eu/concept/history/>
- [15] N. Touran, Molten Salt Reactors.
URL <https://whatisnuclear.com/msr.html>
- [16] S. Technologies, Our Technology.
URL <https://www.seaborg.co/the-reactor>
- [17] V. M. Akse, On fuel drain systems for molten salt reactors, Norwegian University of Science and Technology.
- [18] OpenFoamWiki, Main ContribExamples/AxiSymmetric.
URL https://openfoamwiki.net/index.php/Main_ContribExamples/AxiSymmetric
- [19] F. Khalili, P. Gamage, H. Mansy, Verification of turbulence models for flow in a constricted pipe at low reynolds number, *American Society of Thermal and Fluids Engineers* (2018).
- [20] CFD-Online, Turbulence free-stream BC.
URL https://www.cfd-online.com/Wiki/Turbulence_free-stream_boundary_conditions
- [21] Menter, Florian R., Improved two-equation k-omega turbulence models for aerodynamic flows (1992).
- [22] Simscale, $k - \omega$ SST model.
URL <https://www.simscale.com/docs/content/simulation/model/turbulenceModel/kOmegaSST.html>
- [23] Fangqing Liu, A Thorough Description Of How Wall Functions Are Implemented In OpenFOAM.
URL http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016
- [24] Wikipedia.org, Law of the wall.
URL https://en.wikipedia.org/wiki/Law_of_the_wall
- [25] J. R. Lamarsh, A. J. Baratta, Introduction to Nuclear Engineering, 3rd Edition, Prentice-Hall, 2001 (2001).
- [26] J. R. Lamarsh, Introduction to Nuclear Reactor Theory, 2nd Edition, Addison-Wesley, 1966 (1966).
- [27] M. S. Stacey, Nuclear Reactor Physics, 3rd Edition, Wiley-VCH, 2018 (2018).
- [28] C. V. Subbarao, D. A. N. Divya, P. King, Mechanics of slow draining of large cylindrical tank under gravity, *e-Journal of Science Technology (e-JST)* 39 (101-111) (2019).

Appendix A. OpenFOAM final case

```
/*----- C++ -----*\
=====
\\      / F ield      | OpenFOAM: The Open Source CFD Toolbox
\\      / O peration  | Website:  https://openfoam.org
\\      / A nd        | Version:   7
\\      / M anipulation|
\*-----*/
FoamFile
{
    version      2.0;
    format       ascii;
    class        dictionary;
    location     "system";
    object       controlDict;
}
// * * * * *

application      energyTrans;

startFrom        latestTime;

startTime        0;

stopAt           endTime;

endTime          200;

deltaT           0.0001;

writeControl      adjustableRunTime;

writeInterval     0.5;

purgeWrite        0;

writeFormat       binary;

writePrecision    6;

writeCompression  off;

timeFormat        general;

timePrecision     6;

runTimeModifiable yes;

adjustTimeStep    yes;

maxCo             0.9;//0.5;
maxAlphaCo        0.9;//0.5;

maxDeltaT         1;
```

```

functions
{
    #includeFunc singleGraphECell

    minMax
    {
        type      fieldMinMax;
        libs      ("libfieldFunctionObjects.so");
        writeControl adjustableRunTime;
        writeInterval 0.5;
        fields      (T);
    }
}
// *****

```

```

/*-----* C++ *-----*/
|=====|
|  \ \  /  | F ield      | OpenFOAM: The Open Source CFD Toolbox
|  \ \  /  | O peration | Version: 3.0.1
|  \ \  /  | A nd       | Web: www.OpenFOAM.org
|  \ \  /  | M anipulation|
|=====|
/*-----*/
FoamFile
{
    version      2.0;
    format       ascii;
    class        dictionary;
    object       blockMeshDict;
}
// * * * * *

convertToMeters 0.01;

rad          #calc "degToRad(2.5)";

pipe_x       #calc "5*cos($rad)";
pipe_zp      #calc "5*sin($rad)";
pipe_zn      #calc "-5*sin($rad)";

x            #calc "100*cos($rad)";
zp           #calc "100*sin($rad)";
zn           #calc "-100*sin($rad)";
/*
out_x        #calc x/10;
out_zp       #calc zp/10;
out_zn       #calc zn/10;
*/

vertices
(
    (0 0 0) //0
    ($pipe_x 0 $pipe_zp) //1
    ($pipe_x 100 $pipe_zp) //2
    (0 100 0) //3
    ($pipe_x 0 $pipe_zn) //4
    ($pipe_x 100 $pipe_zn) //5
    ($pipe_x 300 $pipe_zp) //6
    (0 300 0) //7
    ($pipe_x 300 $pipe_zn) //8
    ($x 100 $zp) //9
    ($x 300 $zp) //10
    ($x 100 $zn) //11
    ($x 300 $zn) //12
    ($pipe_x 330 $pipe_zp) //13
    (0 330 0) //14
    ($pipe_x 330 $pipe_zn) //15
    ($x 330 $zp) //16
    ($x 330 $zn) //17
);

```

```

blocks
(
    //pipe
    hex (0 1 4 0 3 2 5 3) waterZone (8 1 80) simpleGrading (0.32 1 1)
    //water over pipe
    hex (3 2 5 3 7 6 8 7) waterZone (8 1 80) simpleGrading (0.32 1 1)
    //water in rest of tank
    hex (2 9 11 5 6 10 12 8) waterZone (40 1 80) simpleGrading (11.8 1 1)
    //hex (2 9 11 5 6 10 12 8) waterZone (35 1 75) simpleGrading (1 1 1)
    //air over pipe
    hex (7 6 8 7 14 13 15 14) (8 1 8) simpleGrading (0.32 1 1)
    //air over rest of tank
    hex (6 10 12 8 13 16 17 15) (40 1 8) simpleGrading (11.8 1 1)
);

edges
(
    arc 1 4 (5 0 0)
    arc 2 5 (5 100 0)
    arc 6 8 (5 300 0)
    arc 10 12 (100 300 0)
    arc 16 17 (100 330 0)
);

boundary
(
    axis
    {
        type empty;
        faces
        (
            (0 3 3 0)
            (3 7 7 3)
            (7 14 14 7)
        );
    }
    pipe
    {
        type wall;
        faces
        (
            (1 4 5 2) //pipe
        );
    }
    lower
    {
        type wall;
        faces
        (
            (2 5 11 9) //lower
        );
    }
}

```

```

outer
{
    type wall;
    faces
    (
        (9 11 12 10)//outer
        //(13 15 14 13) //
        (10 12 17 16)
    );
}

```

```

rightWall
{
    //type patch;
    type wedge;
    faces
    (
        (0 1 2 3)
        (3 2 6 7)
        (2 9 10 6)
        (7 6 13 14)
        (6 10 16 13)
    );
}

```

```

leftWall
{
    //type patch;
    type wedge;
    faces
    (
        (0 3 5 4)
        (3 7 8 5)
        (5 8 12 11)
        (8 7 14 15)
        (8 15 17 12)
    );
}

```

```

outlet
{
    type patch;
    faces
    (
        (0 4 1 0)
    );
}

```

```

atmosphere //top
{
    type patch;
    faces
    (
        (14 13 15 14)
        (13 16 17 15)
    );
}

```



```
        );  
    }  
  
);  
  
mergePatchPairs  
(  
);  
  
// ***** //
```



```

/*----- C++ -----*/
|=====|
|  \ \  /  | F ield      | OpenFOAM: The Open Source CFD Toolbox
|  \ \  /  | O peration | Version: 3.0.1
|  \ \  /  | A nd       | Web: www.OpenFOAM.org
|  \ \  /  | M anipulation |
|=====|
/*-----*/
FoamFile
{
    version      2.0;
    format       ascii;
    class        dictionary;
    location     "system";
    object       setFieldsDict;
}
// *****

defaultFieldValues
(
    volScalarFieldValue alpha.water 0
);

regions
(
    zoneToCell
    {
        name "waterZone";
        fieldValues
        (
            volScalarFieldValue alpha.water 1
        );
    }
);

// *****

```

Appendix B. OpenFOAM solver

```

{
    fvScalarMatrix TEqn
    (
        fvm::ddt(rho, T) + fvm::div(rhoPhi, T) - fvm::Sp(contErr, T)
        - fvm::laplacian(turbulence.alphaEff(), T)
        + (
            fvc::div(fvc::absolute(phi, U), p)()() // - contErr/rho*p
            + (fvc::ddt(rho, K) + fvc::div(rhoPhi, K))()() - contErr*K
        )
        * (
            alpha1()/mixture.thermo1().Cv()()
            + alpha2()/mixture.thermo2().Cv()()
        )
    )
    ==
    fvOptions(rho, T)
    + Source
);
TEqn.relax();

fvOptions.constrain(TEqn);

TEqn.solve();

fvOptions.correct(T);

mixture.correctThermo();
mixture.correct();
}

```



```

// * * * * *
int main(int argc, char *argv[])
{
    #include "postProcess.H"

    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createDynamicFvMesh.H"
    #include "initContinuityErrs.H"
    #include "createDyMControls.H"
    #include "createFields.H"
    #include "CourantNo.H"
    #include "setInitialDeltaT.H"
    #include "createUfIfPresent.H"

    volScalarField& p = mixture.p();
    volScalarField& T = mixture.T();
    const volScalarField& psi1 = mixture.thermo1().psi();
    const volScalarField& psi2 = mixture.thermo2().psi();

    // * * * * *
    Info<< "\nStarting time loop\n" << endl;

    while (runTime.run())
    {
        #include "readDyMControls.H"

        // Store divU from the previous mesh so that it can be mapped
        // and used in correctPhi to ensure the corrected phi has the
        // same divergence
        volScalarField divU("divU0", fvc::div(fvc::absolute(phi, U)));

        if (LTS)
        {
            #include "setRDeltaT.H"
        }
        else
        {
            #include "CourantNo.H"
            #include "alphaCourantNo.H"
            #include "setDeltaT.H"
        }

        runTime++;

        Info<< "Time = " << runTime.timeName() << nl << endl;

        forAll(Source(), i)
        {
            // if (alpha1()[i] > 0.9999)
            // {
            if (runTime.time().value() > 0.1)
            {
                if (runTime.time().value() < 10)

```



```

    {
        Source.ref()[i] =alpha1()[i]*(250e6*5e-3*12.05*(Foam::pow
            ((runTime.time().value()),-0.0639)-Foam::pow(8.64e6 +
            (runTime.time().value()),-0.0639)))/(6.28*1360);
    }
    else if(runTime.time().value()<150)
    {
        Source.ref()[i] =alpha1()[i]*(250e6*5e-3*15.31*(Foam::pow
            ((runTime.time().value()),-0.1807)-Foam::pow(8.64e6 +
            (runTime.time().value()),-0.1807)))/(6.28*1360);
    }
    else
    {
        Source.ref()[i] =alpha1()[i]*(250e6*5e-3*27.43*(Foam::pow
            ((runTime.time().value()),-0.2962)-Foam::pow(8.64e6 +
            (runTime.time().value()),-0.2962)))/(6.28*1360);
    }
}
//}
else
{
    Source.ref()[i] = 0;
}
}
// — Pressure-velocity PIMPLE corrector loop
while (pimple.loop())
{
    if (pimple.firstPimpleIter() || moveMeshOuterCorrectors)
    {
        mesh.update();

        if (mesh.changing())
        {
            gh = (g & mesh.C()) - ghRef;
            ghf = (g & mesh.Cf()) - ghRef;

            MRF.update();

            if (correctPhi)
            {
                // Calculate absolute flux
                // from the mapped surface velocity
                phi = mesh.Sf() & Uf();

                #include "correctPhi.H"

                // Make the fluxes relative to the mesh motion
                fvc::makeRelative(phi, U);
            }

            mixture.correct();

            if (checkMeshCourantNo)
            {
                #include "meshCourantNo.H"
            }
        }
    }
}

```

```

    }

    divU = fvc::div(fvc::absolute(phi, U));

    #include "alphaControls.H"
    #include "compressibleAlphaEqnSubCycle.H"

    turbulence.correctPhasePhi();

    #include "UEqn.H"
    #include "TEqn.H"

    // — Pressure corrector loop
    while (pimple.correct())
    {
        #include "pEqn.H"
    }

    if (pimple.turbCorr())
    {
        turbulence.correct();
    }

}

runTime.write();

Info<< "ExecutionTime = "
    << runTime.elapsedCpuTime()
    << " s\n\n" << endl;
}

Info<< "End\n" << endl;

return 0;
}

// ***** //
```

```

#include "createRDeltaT.H"

Info<< "Reading field p_rgh\n" << endl;
volScalarField p_rgh
(
    IOobject
    (
        "p_rgh",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

Info<< "Reading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

#include "createPhi.H"

Info<< "Constructing twoPhaseMixtureThermo\n" << endl;
twoPhaseMixtureThermo mixture(U, phi);

volScalarField& alpha1(mixture.alpha1());

volScalarField& alpha2(mixture.alpha2());

Info<< "Reading alpha2\n" << endl;

const volScalarField& rho1 = mixture.thermo1().rho();
const volScalarField& rho2 = mixture.thermo2().rho();

volScalarField rho
(
    IOobject
    (
        "rho",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),

```

```

        alpha1*rho1 + alpha2*rho2
    );

dimensionedScalar pMin
(
    "pMin",
    dimPressure,
    mixture
);

mesh.setFluxRequired(p_rgh.name());
mesh.setFluxRequired(alpha1.name());

#include "readGravitationalAcceleration.H"
#include "readhRef.H"
#include "gh.H"

// Mass flux
// Initialisation does not matter because rhoPhi is reset after the
// alpha1 solution before it is used in the U equation.
surfaceScalarField rhoPhi
(
    IOobject
    (
        "rhoPhi",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    fvc::interpolate(rho)*phi
);

volScalarField dgdt(alpha1*fvc::div(phi));

#include "createAlphaFluxes.H"

volScalarField Source
(
    IOobject
    (
        "Source",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh,

```

```

        dimensionedScalar("Source", dimensionSet(1,-3,-1,1,0,0,0),0)
    );


// Construct compressible turbulence model
compressibleInterPhaseTransportModel turbulence
(
    rho,
    U,
    phi,
    rhoPhi,
    alphaPhi10,
    mixture
);

Info<< "Creating field kinetic energy K\n" << endl;
volScalarField K("K", 0.5*magSqr(U));

#include "createMRF.H"
#include "createFvOptions.H"

```

Appendix C. OpenFOAM src



F i e l d

O p e r a t i o n

A n d

M a n i p u l a t i o n

OpenFOAM: The Open Source CFD Toolbox

Website: <https://openfoam.org>

Copyright (C) 2012–2019 OpenFOAM Foundation

License

This file is part of OpenFOAM.

OpenFOAM is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

OpenFOAM is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with OpenFOAM. If not, see <http://www.gnu.org/licenses/>.

```
#include "rhoCustom.H"
```

```
// * * * * * Private Member Functions * * * * *
```

```
template<class Specie>
inline Foam::rhoCustom<Specie>::rhoCustom
(
    const Specie& sp,
    const scalar R,
    const scalar rho0
)
:
    Specie(sp),
    R_(R),
    rho0_(rho0)
{}

```

```
// * * * * * Constructors * * * * *
```

```
template<class Specie>
inline Foam::rhoCustom<Specie>::rhoCustom
(
    const word& name,
    const rhoCustom<Specie>& pf
)
:
    Specie(name, pf),
    R_(pf.R_),
```



```

    rho0_(pf.rho0_)
{}

template<class Specie>
inline Foam::autoPtr<Foam::rhoCustom<Specie>>
Foam::rhoCustom<Specie>::clone() const
{
    return autoPtr<rhoCustom<Specie>>(new rhoCustom<Specie>(*this));
}

template<class Specie>
inline Foam::autoPtr<Foam::rhoCustom<Specie>>
Foam::rhoCustom<Specie>::New
(
    const dictionary& dict
)
{
    return autoPtr<rhoCustom<Specie>>(new rhoCustom<Specie>(dict));
}

// * * * * * Member Functions * * * * * //

template<class Specie>
inline Foam::scalar Foam::rhoCustom<Specie>::R() const
{
    return R_;
}

template<class Specie>
inline Foam::scalar Foam::rhoCustom<Specie>::rho(scalar p, scalar T) const
{
    return 3606.4-0.201*T;
}

template<class Specie>
inline Foam::scalar Foam::rhoCustom<Specie>::H(scalar p, scalar T) const
{
    return p/rho(p, T) - Pstd/rho(Pstd, T);
}

template<class Specie>
inline Foam::scalar Foam::rhoCustom<Specie>::Cp(scalar p, scalar T) const
{
    return 1360;
}

template<class Specie>
inline Foam::scalar Foam::rhoCustom<Specie>::E(scalar p, scalar T) const
{
    return 0;
}

```

```

}

template<class Specie>
inline Foam::scalar Foam::rhoCustom<Specie>::Cv(scalar p, scalar T) const
{
    return 0;
}

template<class Specie>
inline Foam::scalar Foam::rhoCustom<Specie>::S(scalar p, scalar T) const
{
    return 0;
}

template<class Specie>
inline Foam::scalar Foam::rhoCustom<Specie>::psi(scalar p, scalar T) const
{
    return 0;
}

template<class Specie>
inline Foam::scalar Foam::rhoCustom<Specie>::Z(scalar p, scalar T) const
{
    return 0;
}

template<class Specie>
inline Foam::scalar Foam::rhoCustom<Specie>::CpMCv(scalar p, scalar T) const
{
    return 0;
}

// * * * * * Member Operators * * * * * //

template<class Specie>
inline void Foam::rhoCustom<Specie>::operator+=
(
    const rhoCustom<Specie>& pf
)
{
    scalar Y1 = this->Y();
    Specie::operator+=(pf);

    if (mag(this->Y()) > small)
    {
        Y1 /= this->Y();
        const scalar Y2 = pf.Y()/this->Y();

        R_ = 1.0/(Y1/R_ + Y2/pf.R_);
        rho0_ = Y1*rho0_ + Y2*pf.rho0_;
    }
}

```

```

}

template<class Specie>
inline void Foam::rhoCustom<Specie>::operator*=(const scalar s)
{
    Specie::operator*=(s);
}

// * * * * * Friend Operators * * * * * //

template<class Specie>
inline Foam::rhoCustom<Specie> Foam::operator+
(
    const rhoCustom<Specie>& pf1,
    const rhoCustom<Specie>& pf2
)
{
    Specie sp
    (
        static_cast<const Specie>(pf1)
        + static_cast<const Specie>(pf2)
    );

    if (mag(sp.Y()) < small)
    {
        return rhoCustom<Specie>
        (
            sp,
            pf1.R_,
            pf1.rho0_
        );
    }
    else
    {
        const scalar Y1 = pf1.Y()/sp.Y();
        const scalar Y2 = pf2.Y()/sp.Y();

        return rhoCustom<Specie>
        (
            sp,
            1.0/(Y1/pf1.R_ + Y2/pf2.R_),
            Y1*pf1.rho0_ + Y2*pf2.rho0_
        );
    }
}

template<class Specie>
inline Foam::rhoCustom<Specie> Foam::operator*
(
    const scalar s,
    const rhoCustom<Specie>& pf
)
{
    return rhoCustom<Specie>

```

```

(
    s*static_cast<const Specie>(pf),
    pf.R_,
    pf.rho0_
);
}

template<class Specie>
inline Foam::rhoCustom<Specie> Foam::operator==
(
    const rhoCustom<Specie>& pf1,
    const rhoCustom<Specie>& pf2
)
{
    Specie sp
    (
        static_cast<const Specie>(pf1)
        == static_cast<const Specie>(pf2)
    );

    if (mag(sp.Y()) < small)
    {
        return rhoCustom<Specie>
        (
            sp,
            pf1.R_,
            pf1.rho0_
        );
    }
    else
    {
        const scalar Y1 = pf1.Y()/sp.Y();
        const scalar Y2 = pf2.Y()/sp.Y();
        const scalar oneByR = Y2/pf2.R_ - Y1/pf1.R_;

        return rhoCustom<Specie>
        (
            sp,
            mag(oneByR) < small ? great : 1/oneByR,
            Y2*pf2.rho0_ - Y1*pf1.rho0_
        );
    }
}

// ***** //
```



```

inline Foam::autoPtr<Foam::customTransport<Thermo>>
Foam::customTransport<Thermo>::clone() const
{
    return autoPtr<customTransport<Thermo>>
    (
        new customTransport<Thermo>(*this)
    );
}

template<class Thermo>
inline Foam::autoPtr<Foam::customTransport<Thermo>>
Foam::customTransport<Thermo>::New
(
    const dictionary& dict
)
{
    return autoPtr<customTransport<Thermo>>
    (
        new customTransport<Thermo>(dict)
    );
}

// * * * * * Member Functions * * * * * //

template<class Thermo>
inline Foam::scalar Foam::customTransport<Thermo>::mu
(
    const scalar p,
    const scalar T
) const
{
    return 7.7e-5*exp(4444.444/T);
}

template<class Thermo>
inline Foam::scalar Foam::customTransport<Thermo>::kappa
(
    const scalar p,
    const scalar T
) const
{
    return 1.19;//this->kappa(p, T);
}

template<class Thermo>
inline Foam::scalar Foam::customTransport<Thermo>::alphah
(
    const scalar p,
    const scalar T
) const
{
    return kappa(p, T)/1360;
}

```



```
// * * * * * Member Operators * * * * * //
```

```
template<class Thermo>
inline void Foam::customTransport<Thermo>::operator+=
(
    const customTransport<Thermo>& st
)
{
    scalar Y1 = this->Y();

    Thermo::operator+=(st);

    if (mag(this->Y()) > small)
    {
        Y1 /= this->Y();
        scalar Y2 = st.Y()/this->Y();

        mu_ = Y1*mu_ + Y2*st.mu_;
        rPr_ = 1.0/(Y1/rPr_ + Y2/st.rPr_);
    }
}
```

```
template<class Thermo>
inline void Foam::customTransport<Thermo>::operator*=
(
    const scalar s
)
{
    Thermo::operator*=(s);
}
```

```
// * * * * * Friend Operators * * * * * //
```

```
template<class Thermo>
inline Foam::customTransport<Thermo> Foam::operator+
(
    const customTransport<Thermo>& ct1,
    const customTransport<Thermo>& ct2
)
{
    Thermo t
    (
        static_cast<const Thermo&>(ct1) + static_cast<const Thermo&>(ct2)
    );

    if (mag(t.Y()) < small)
    {
        return customTransport<Thermo>
        (
            t,
            0,
            ct1.rPr_
        );
    }
}
```

```

    }
    else
    {
        scalar Y1 = ct1.Y()/t.Y();
        scalar Y2 = ct2.Y()/t.Y();

        return customTransport<Thermo>
        (
            t,
            Y1*ct1.mu_ + Y2*ct2.mu_,
            1.0/(Y1/ct1.rPr_ + Y2/ct2.rPr_)
        );
    }
}

template<class Thermo>
inline Foam::customTransport<Thermo> Foam::operator*
(
    const scalar s,
    const customTransport<Thermo>& ct
)
{
    return customTransport<Thermo>
    (
        s*static_cast<const Thermo>(ct),
        ct.mu_,
        1.0/ct.rPr_
    );
}

// ***** //
```