11-10-2021

Simulation of Bulk Evaporation and Condensation in Cryogenic Propellant Tanks using the Energy of Fluid Method

Elijah Gasmen

SIMULATION OF BULK EVAPORATION AND CONDENSATION IN CRYOGENIC PROPELLANT TANKS USING THE ENERGY OF FLUID METHOD

by

Elijah Gasmen

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Major: Mechanical Engineering

The University of Memphis

December 2021

ACKNOWLEDGEMENTS

The research presented is supported by the University of Memphis Department of Mechanical Engineering and the Tennessee Space Grant Consortium.

ABSTRACT

The design of cryogenic propellant storage systems for long duration space missions relies on accurate prediction of tank self-pressurization. Incident solar radiation heats the cryogenic liquids in the tank over time, vaporizing the cryogenic liquid. As the liquid vaporizes, the tank pressure increases. The objective of the current research is to develop a finite volume based Computational Fluid Dynamic (CFD) model of tank pressurization in reduced gravity using an Energy of Fluid (EOF) approach. A commercially available CFD model is significantly enhanced to include the EOF method, which will solve the energy equation in terms of internal energy. Model validation results are presented which include a comparison of temperature and pressure predictions to the data collected during the terrestrial experiments performed by Aydelott and the low gravity experiment conducted onboard the Saturn IB AS203 tank.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

a	Influence coefficient
Bo	Bond number, $\frac{\Delta\rho g R^2}{\sigma}$
c_v	Specific heat at constant volume, J/kg-K
D	Diffusive flux
ELF	Evaporated liquid fraction
E	Total heat, specific internal energy, J/kg
e	Sensible heat, J/kg
Δe	Latent heat, J/kg
F	Convective flux
f	Phase fraction
g	Gravitational acceleration, m/s^2
G	Gravity level
k	Thermal conductivity, W/m-K
L	Length, m
m	Mass, kg
P	Pressure, Pa
Q	Heat rate, W
q	Heat flux, W/m^2
R	Radius, m
Ra	Rayleigh number, $\frac{g\beta\Delta T L^3}{\nu\alpha}$
S	Source term
T	Temperature, K
t	Time, sec
Δt	Timestep size, sec
u	Velocity, m/s
V	Volume, m^3
Y	Phase front location, m
Δy	Vertical cell length
α	Thermal diffusivity, m^2/s
β	Thermal expansion coefficient, 1/K
Γ	Diffusion coefficient
ν	Kinematic viscosity, m^2/s
ρ	Density, kg/m^3
σ	Surface tension, N/m
ϕ	General variable
χ	Relaxation factor

Superscripts

$*$	Previous iteration
0	Previous timestep
n	Time step

Subscripts

P	Current cell
nb	Neighboring cell
g	Vapor
l	Liquid
ref	Reference value
$face$	Face-centered value
cell	Cell-centered value

INTRODUCTION

In long duration space missions, cryogenic liquid propellant tanks experience heat leaks from incident solar radiation. The heat evaporates liquid propellant, which causes a rise in tank pressure through the process called self-pressurization. This effect is undesirable for several reasons. Vaporization of liquid reduces the amount of valuable propellant available for the rocket propulsion systems. An immense pressure buildup in the tank can result in rupture if left uncontrolled [1].

The prediction of the pressure rise rate is an important factor in the design of the cryogenic propellant tanks. This prediction is also important for specifying the propellant requirements and any orbital tank venting requirements. Numerical models predicting self-pressurization in cryogenic tanks have been under continuous development. These models range from simplified lumped parameter models to more complex finite-volume-based methods.

In this research, a numerical model is developed to predict the self-pressurization process by utilizing commercial computational fluid dynamics (CFD) software based on the finite volume method. The software is significantly enhanced to model heat transfer and phase change using the Energy of Fluid (EOF) approach, which uses an internal energy formulation for the energy equation. Experimental cases are used to validate the accuracy of the model. The results of the validation studies show that the current model can reproduce the outcome of a low gravity experiment.

Background

i. Low Gravity Orbital Fuel Tank Experiment

A case investigated to validate the computational model is the closed fuel tank experiment conducted in the S-IVB stage of vehicle AS-203 in 1966 [1]. The vehicle, shown on Figure 1, was launched into orbit and conducted a series of experiments to validate numerical models and the performance of the vehicle for lunar travel. The objective of the experiment was to determine the pressure rise of the closed liquid hydrogen fuel tank in orbit.

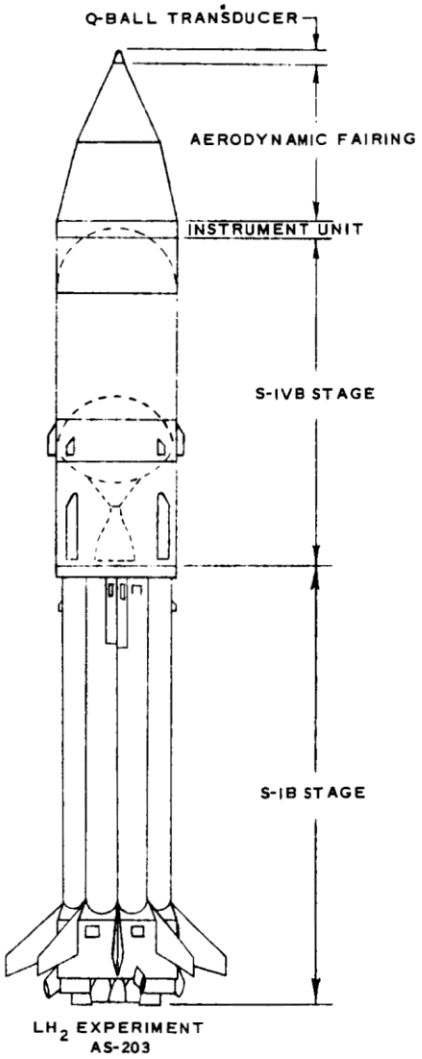


Figure 1 Vehicle AS-203 stages and components.

At the beginning of the experiment, all valves connected to the LH2 tank were closed, and the vehicle continued to accelerate to keep the liquid settled at the aft end. The vehicle experienced apparent gravity levels ranging from 3.5E-4 to 7.8E-5, shown on Figure 2. These values are normalized based on the normal gravity of Earth.

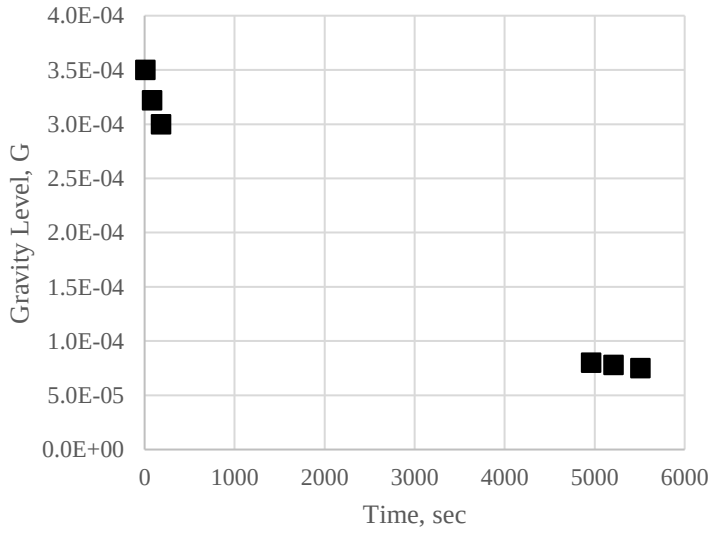


Figure 2 Vehicle acceleration data during experiment.

An approximate LH2 mass of 7257 kg was present in the tank at a pressure of 85,495 Pa. The tank self-pressurized as heat from solar radiation caused the hydrogen to evaporate and the temperature to rise. Tank pressure data was sent from the vehicle to tracking stations via telemetry. Pressure data was not received during the middle of the experiment due to signal loss. The pressure at the end of the experiment was 259,932 Pa. With an experiment duration of 5360 sec, the corresponding pressure rise rate was approximately 32.5 Pa/s. Shortly after the conclusion of the experiment, the LH2 tank ruptured due the immense pressure gradient located at the common bulkhead between the LH2 and LOX tanks.

This experiment was the subject of several simulation attempts to predict the pressure rise. The recorded pressure data from the experiment is shown on Figure 3 as well as the results from each of the following cases. Ward employed a simplified thermodynamic model of the

tank, which separated the fluid domain into a homogenous ullage section and a homogenous liquid section [1]. The heat rates for the sections were determined using the temperature distributions and energy balances of the ullage and liquid regions. The energy balance of the ullage region reported a boil-off value of approximately 113 kg of vaporized liquid. An initial pressure rise model by Ward ignored the heat input into the ullage space, and the heat only flowed into the liquid space. This model severely underpredicted the pressure rise rate at 6.13 Pa/s. Ward updated the model to feature heating rates for both the ullage and liquid regions. This updated model predicted an improved pressure rise that was slightly lower than the experiment at 27.7 Pa/s. Ward concluded that accurate prediction of the pressure rise required both ullage and liquid heating rates as well as modeling of thermal stratification within the tank.

Like Ward, Bradshaw used a model that depicted the tank as an ullage region and a liquid region [2]. Unlike Ward, Bradshaw determined the heating rates based on the fuel tank geometry, tank wall absorptivity, incident solar radiation, and changes in the fluid properties. Bradshaw used these heat rates and two programs to predict the pressure rise. The first program, REPORTER, predicted a higher pressure near the beginning of the experiment and a lower pressure near the end. This program reported no evaporation in the liquid and lacked stratification in the model. The second program, P3542, did include evaporation in the liquid. Bradshaw investigated two cases of slightly differing ullage heat inputs with the P3542 program. The results show that the pressure rise rate was highly dependent upon the heat added to the system. The first case had a heat input of 52,922 kJ, slightly overpredicted the pressure at 262,690 Pa, and predicted a boil-off of 14.51 kg. The second case had a heat input of 58,872 kJ, overpredicted the pressure at 283,375 Pa, and predicted a boil-off of 16.33 kg. While these

programs predicted the pressure rate within the desired tolerance, thermal stratification was still absent in the models.

A more recent study by Winter used the commercially available CFD software Fluent, which was enhanced by the EOF method to model the pressure rise [3]. The energy equation implemented into the software featured transient conduction phase change but lacked fluid flow and convection heat transfer. This conduction model adequately predicted an end pressure of 239,459 Pa. However, several aspects of this model require improvement with respect to other validation criteria. A net condensation was observed within the tank, unlike in the previous models mentioned. A temperature map based on data at the end of the experiment is shown on Figure 4 [1]. The map shows isotherms graduated along the axial direction of the tank, which cannot be replicated by a strictly conduction-based model. These additional criteria show that modeling convection heat transfer is required to adequately simulate the fluid within the tank. A Rayleigh number analysis is presented in the Objective section and shows the influence of convection within the tank.

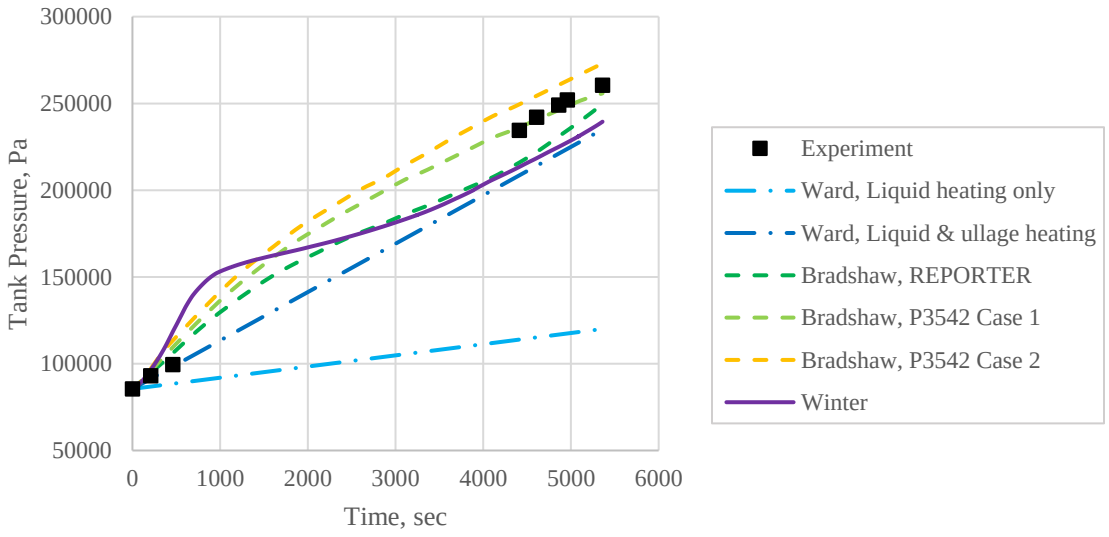


Figure 3 Summary of pressure rise results from AS-203 closed fuel tank experiment and various past simulation works.

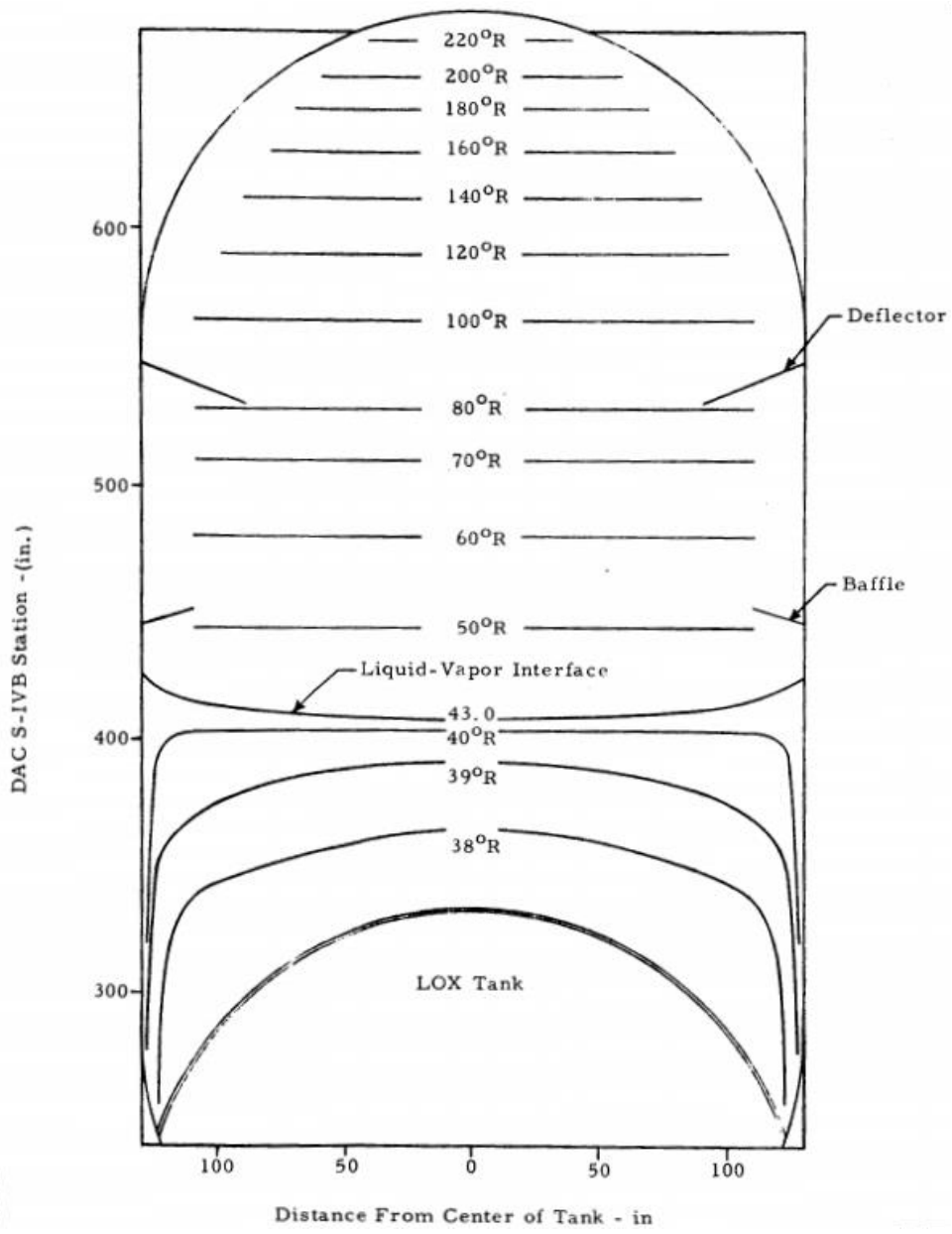


Figure 4 Temperature map at end of closed fuel tank experiment.

ii. Normal Gravity Small Scale Terrestrial Tank Experiment

A second validation case to be investigated was the self-pressurization of a spherical LH2 tank under normal gravity conditions [4]. The tank was significantly smaller than the AS203 fuel tank with a diameter of 23 cm. The experiments were conducted within a vacuum-insulated apparatus cooled by LN2 to mitigate uncontrolled heat leaks. The apparatus was heated using electric heating coils mounted underneath the vacuum insulation and outside of the sphere containing the LH2. This series of experiments included many cases of differing heating configurations, wall heat rates, and initial fill levels. A process of elimination using preferred experiment conditions prioritizes which one of these runs will be simulated by the model. The chosen case is run T16, which has a distinct liquid-vapor interface, a top heating configuration, a 50% initial fill level, and a low heat input.

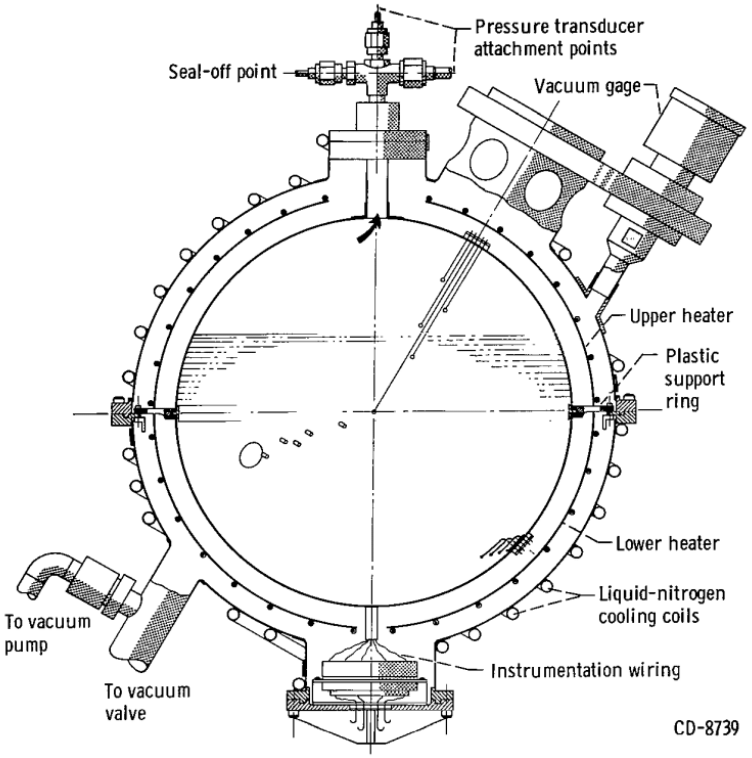


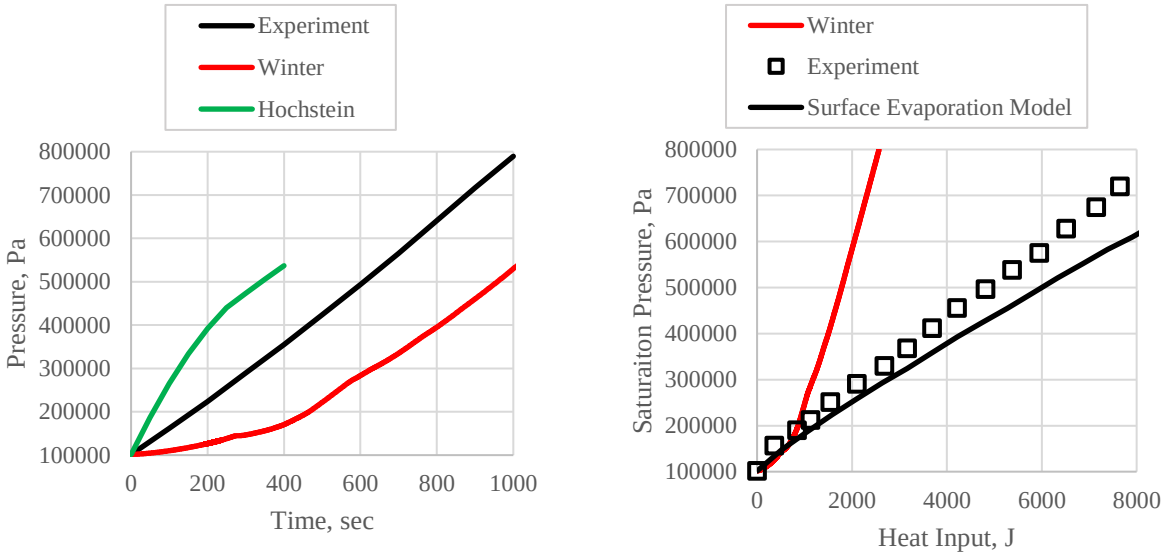
Figure 5 Apparatus for normal gravity small scale terrestrial LH2 tank self-pressurization experiment.

Three investigators attempted to recreate the pressure rise, discussed below. The pressure rise results are shown with respect to different axes. Figure 6a shows a comparison of the results with respect to time. Figure 6b shows a similar comparison but with respect to the overall heat input. The first investigator, Aydelott, employed a theoretical surface evaporation model, which assumed that all heat entering the tank was put into the evaporation of liquid. The two phases were assumed to remain at saturation conditions. The model underpredicted the pressure rise, and this difference increased as more heat is added.

Hochstein [5] simulated this case using the Solution Algorithm enhanced by Energy Calculations for Liquid Propellants in a Space Environment (SOLA-ECLIPSE) code. The model used a Volume of Fluid (VOF) method to distinguish between the two fluid phases. The liquid region was modeled using a finite volume method. The vapor region was modeled as a single node connected to the tank walls and to each free surface cell. For the sake of simplicity, the convection term is not present in the energy equation for the liquid region. Therefore, an effective thermal conductivity was used to describe the heat transfer. The effective conductivity consolidated the convection and conduction modes of heat transfer into a purely conduction term in the energy equation. The pressure was calculated from the vapor energy and density using a thermodynamic model based on the integration of the Maxwell equations. In addition to modeling this specific experiment run, Hochstein also simulated a case with a uniform heating configuration and a case with a bottom heating configuration. Despite showing excellent correlations for those two cases, the resulting pressure for the top heating case was overpredicted near the start of the simulation. After this initial overprediction, the slope of the pressure rise appeared parallel to that of the experiment. Hochstein concluded that these discrepancies may be

due to inadequacies in modeling the interface and using the effective conductivity in the liquid region.

This case was also investigated by Winter [3] using similar methods to those described with the low gravity experiment. Inadequacies due to lack of natural convection modeling in the low gravity case are greatly increased in a normal gravity case. The predicted pressure rise with respect to time was much lower than that of the experiment. In contrast, the predicted pressure rise with respect to the heat input was considerably higher than the prediction by Aydelott. Winter concluded that the addition of natural convection is necessary to accurately predict the pressure rise within the tank. A Rayleigh number analysis of this case, presented in the Objective section, shows the influence of convection over conduction as the primary mode of heat transfer.



(a) **(b)**

Figure 6 Comparison of pressure rise predictions with respect to (a) time and (b) heat input.

Energy of Fluid Method

The EOF method derives the energy equation from the first law of thermodynamics with no mechanical work [5]. This reduced form states that the change in internal energy of a closed, rigid thermodynamic system is equal to the heat transfer into the system. The unsteady energy equation with pure conduction and incompressible flow is given as Eq. (1), where E is the specific internal energy of a computational cell [6].

$$\rho \frac{\partial E}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} E) = \nabla \cdot (k \nabla T) \tag{1}$$

This approach is similar in concept to the more popularly used enthalpy formulation. Research shows that the two methods produce the same results in similar computational times [5]. However, the EOF method reduces complexity by omitting a transient pressure term present in the enthalpy formulation. Equation (2) calculates the cell internal energy using the summation of the sensible heat and the latent heat content.

$$E = e + \Delta E = e + f \Delta e \tag{2}$$

From Eqs. (1) and (2), a form of the energy equation that includes the latent heat content is given as

$$\rho \frac{\partial e}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} e) = \nabla \cdot (k \nabla T) - \rho \Delta e \frac{\partial f}{\partial t} - \Delta e \nabla \cdot (\rho \boldsymbol{u} f) \tag{3}$$

The vapor and liquid phases are distinguished using the phase fraction, f. The phase fraction is calculated based on the internal energy of the cell, using the piecewise function shown in Eq. (4).

$$f = \begin{cases} 0 & \mathrm{E} < e_l \\ \dfrac{E - e_l}{e_g - e_l} & e_l < E < e_g \\ 1 & e_g < \mathrm{E} \end{cases} \tag{4}$$

The subscripts g and l denote values of a saturated vapor and saturated liquid, respectively. Consequently, a phase fraction of 1 denotes a vapor; a phase fraction of 0 denotes a liquid. This approach solves the iterative loop closure problem and allows both phases to be solved using a single set of governing equations. A gradual phase transition is created at the interface as a result.

Updating the phase fraction using the internal energy values and latent heat of vaporization is called the E-based update method. This update method is different from the T-based update method, which uses a predetermined temperature interval centered around the saturation temperature to model the phase change interface. Research shows that the E-based method is superior to the T-based method in terms of computational time, stability, and accuracy [7].

Objective

The pressure predictions previously presented used conduction-based heat transfer phase change models. The justification for neglecting convection heat transfer in the models is the lower levels of gravity experienced during orbit. Despite having gravity levels at least three magnitudes lower than normal, the low gravity experiment shows thermal stratification as an effect of natural convection [1]. Under normal gravity, convection is the most influential mode of heat transfer.

The Rayleigh number quantifies the influence between convection and conduction heat transfer. A Rayleigh number analysis, for both cases and for each phase, is shown on Table 1. The values used in the calculations are provided separately in Table 2 using available data at the end of the experiments [9]. The transport properties are calculated at saturation. The Rayleigh number at which the flow transitions from laminar to turbulent natural convection is approximately 1E9 [10]. All flows in the validation cases appear to be either partially or fully turbulent. The dominant mode of heat transfer within these cases is convection.

Table 1 Rayleigh number analysis for validation cases.

Case	**Phase**	**Rayleigh Number** $\mathrm{Ra} = \frac{g\beta\Delta T L^3}{\nu\alpha}$
Low Gravity	Vapor	1.10E12
	Liquid	1.68E11
Normal Gravity	Vapor	5.99E11
	Liquid	5.01E11

Table 2 Values used for Rayleigh number analysis.

Case	Phase	Gravity	Thermal Expansion Coefficient	Temp. Range	Length Scale	Kinematic Viscosity	Thermal Diffusivity
		g, m/s^2	β, K^{-1}	ΔT, K	L, m	ν, m^2/s	α, m^2/s
Low Gravity	Vapor	7.36E-4	8.53E-3	98.3	6.96	4.09E-7	4.62E-7
	Liquid		2.11E-2	2.83	4.20	1.53E-7	1.27E-7
Normal Gravity	Vapor	9.81	6.98E-3	126	0.114	1.84E-7	1.16E-7
	Liquid		3.31E-2	8.48	0.116	1.20E-7	7.15E-8

The objective of this research is to develop a numerical model that sufficiently predicts the pressure rise, fluid flow, and heat transfer within self-pressurizing cryogenic propellant tanks using the EOF method. Results from an existing EOF-based conduction model are used to make improvements and support fluid flow and convection heat transfer. From the walls to the bulk fluid, convection transports substantially more heat than conduction, especially considering the tank geometry and the conditions of the experiment. In addition, effects of thermal stratification can be modeled, which will result in more accurate results. For simplicity purposes, special turbulence models will not be implemented.

The two validation cases are to be studied. The first case is the low gravity experiment conducted in the Saturn IB AS-203 fuel tank. The second case is the normal gravity experiment conducted in the spherical LH2 tank. Results from the simulations are compared to data obtained from experiment to validate the accuracy of the model. With a validated model, future cryogenic propellant tank designs may be simulated for a range of environmental conditions, geometries, and other factors to predict the pressure rise rate and plan countermeasures accordingly.

COMPUTATIONAL METHODS

Commercial CFD software, ANSYS Fluent, is used to simulate the fluid flow and heat transfer [8]. This software uses the finite volume method and upwind differencing to discretize the governing differential equations into linearized algebraic equations. The software solves these equations using the algebraic multigrid (AMG) method. For the current research, all simulations use the pressure-based solver. Pressure-velocity coupling is evaluated using the semi-implicit-method-for-pressure-linked-equations (SIMPLE). For spatial discretization, gradients are evaluated using the least-squares-cell-based method. Pressure is evaluated with the pressure-staggered-option (PRESTO). Momentum and Energy are evaluated with a First Order Upwind method for stability reasons. First order implicit transient formulation is used to progress through time.

Geometry and mesh creation is accomplished using ANSYS Workbench. The program used to create the geometry is ANSYS Design Modeler, and the program used to mesh the geometry is ANSYS Meshing. Step-by-step processes for geometry and mesh creation of the simulated cases are presented in the appendix.

Recording data such as the saturation pressure is executed using Fluent report definitions and report files [9]. This data is then analyzed using Microsoft Excel to create charts, observe trends, and make comparisons. Animations of simulation contour and vector plots are recorded for some of the simulations. Fluent exports these animation frames to files containing the relevant contour/vector data, which can be converted to picture files and videos. During the simulation, these animation frames are updated and displayed in the Fluent graphics window. These animations are essential to visually express how the flow field develops.

Code Implementation

All subroutines involving the EOF update algorithm are contained in a User Defined Function (UDF) source code (can also be referred to as UDF or source code) written in C [10]. Fluent provides header files, such as *udf.h* and *sg.h*, so that the UDF may access built-in custom functions. The UDF is used to modify flow variables at different points in the solution procedure.

In each source code, there is a series of User Defined Memory (UDM) variables, where each one corresponds to a vital flow variable. Such variables include the current and previous timestep phase fraction and cell energies. The UDM variables are stored at the cell center and can be used at any time during the simulation. Some flow variables, such as the masses and saturation properties, are independent of what cell they are stored in. For the sake of computational efficiency, these variables are stored as global C variables. The conditions of each case to be investigated are different. Therefore, each case uses a different source code. The source codes for each of the verification and validation cases are provided in the appendices along with tutorials to set up each case in Fluent.

Parallelization of the source code is important for decreasing the computational time. Cell loops that involve summations are bounded by the *begin_c_loop_int* environment. As opposed to the regular *begin_c_loop*, the *_int* suffix prohibits the node processes from double counting overlapping cells between partitions. After each cell loop ends, the function PRF_GRSUM1 is used to sum and synchronize desired variables across all nodes. With a parallelized UDF, the computational speed can be increased proportionally to the number of processes assigned to the Fluent program.

Figure 7 shows a flow chart of the solution procedure including the processes that the UDF implements. For conciseness, the subprocesses for the adjust and execute at end functions

are presented separately in Figure 8 and Figure 9, respectively. The following sections discuss these processes, which the flow charts give a visual guide for when they occur in the simulation.

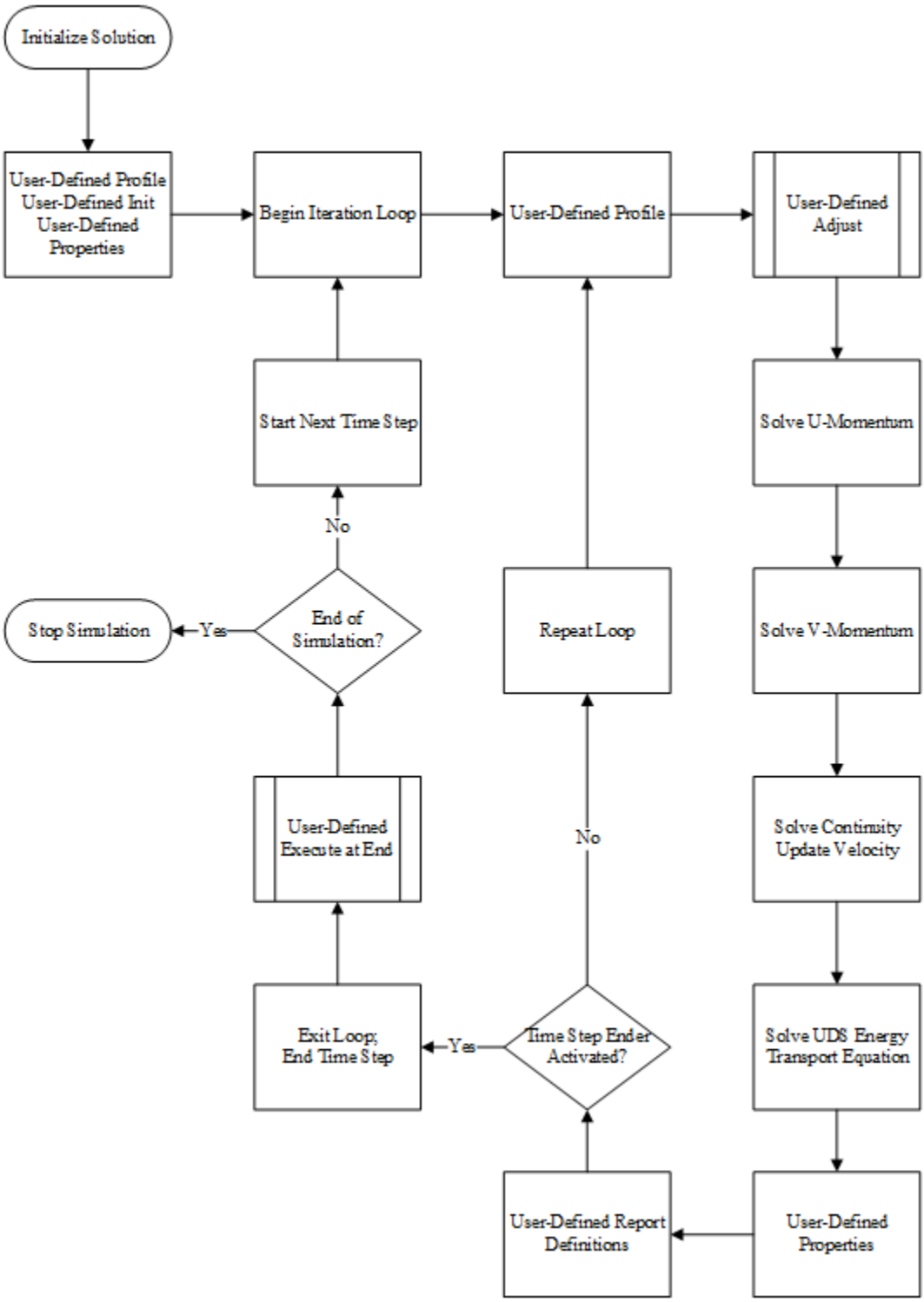


Figure 7 Flow chart of solution procedure.

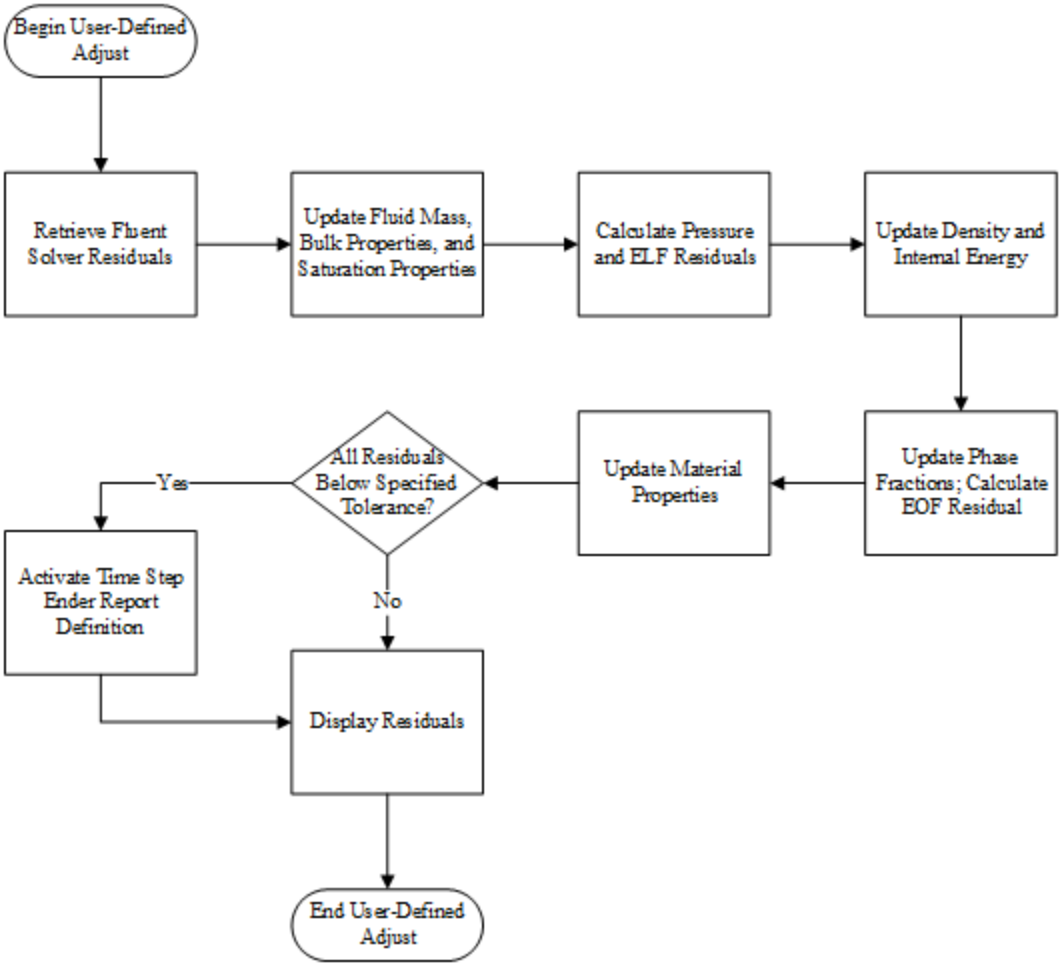


Figure 8 Flow chart of user-defined adjust function.

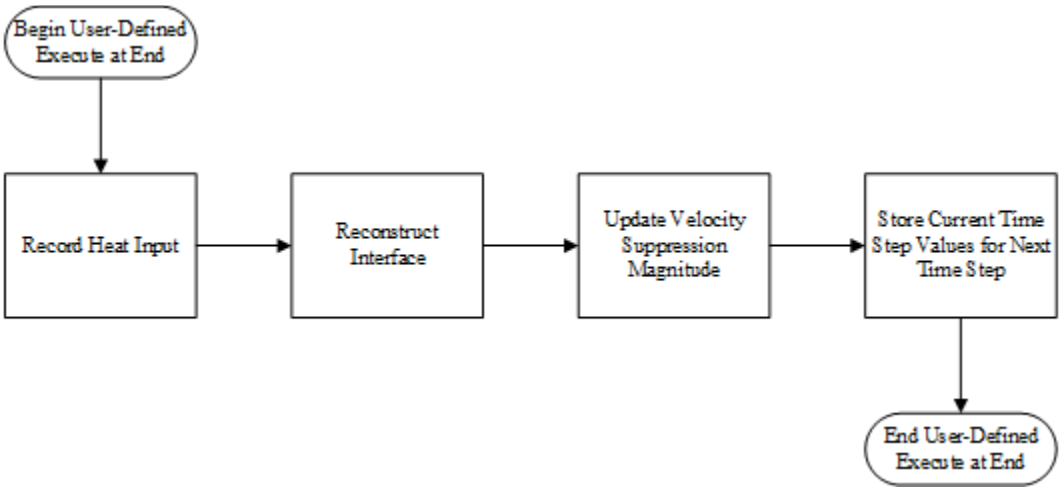


Figure 9 Flow chart of user-defined execute at end function.

Material Properties

The thermophysical properties of parahydrogen are used for the LH2 in the tank. The Maxwell relations form the basis of the thermodynamic model used to obtain the material properties [12]. A 32-term modified Benedict-Webb-Rubin equation of state calculates the pressure as a function of the temperature and density [13, 14]. Reynolds provides a comprehensive list of integrations, curve fits, and coefficients to accurately calculate the properties of hydrogen and other fluids [13]. An existing FORTRAN code based on these equations is obtained from a report by Clark [16]. The code uses an input of temperature and specific volume and gives the combined output for the pressure, internal energy, enthalpy, entropy, saturated liquid density, saturation pressure, and derivative of saturation pressure with respect to saturation temperature. This code is translated to C to interface with the Fluent UDF and is split into functions that output one or two properties instead of the entire set.

There are cases where the output is known, and an input is the desired property. Such is the case with the density. As previously mentioned, the equation of state calculates pressure from temperature and density. To find the density with a known pressure, a root-finding scheme is implemented with the equation of state to iteratively calculate the correct density value that outputs the given pressure. This root-finding scheme is also used to find the saturation temperature with a known saturation pressure. A major disadvantage with this scheme is that the property is found through iteration, which increases the computational time, potentially by an extreme amount. Therefore, use of the root-finding scheme is limited to only the saturation temperature and the superheated vapor density. To further aid convergence, the initial guess to calculate each property is that of the previous Fluent iteration.

The liquid density and saturated vapor density are assumed to be functions of temperature only. The compressed liquid density is set to the saturated liquid density for a given cell temperature. Younglove [15] provides the curve fit equations and coefficients for these saturation densities in terms of the saturation temperature.

The thermal conductivity and dynamic viscosity do not change significantly between short ranges of temperature and pressure. These properties are calculated using sets of linear interpolations between data points from NIST [16]. The data tables for these properties are constructed according to temperature and pressure axes. The temperature axes are split into a saturated axis and a superheated axis. The compressed liquid tables assume dependence with only temperature and are tabulated corresponding to the saturation temperature axis values. The superheated vapor tables are tabulated with respect to the superheated temperature axis and the pressure axis. The data tables and look-up functions are assembled in a separate source code, included in Appendix K.

The material properties for a mixed phase cell are calculated using a volume-weighted average method based on the phase fraction [16]. Equation (5) shows the linear interpolation scheme for a general cell-centered property ϕ weighted by the phase fraction f. The subscripts of g and l denote values of a saturated vapor and liquid, respectively.

$$\phi = f\phi_g + (1 - f)\phi_l \tag{5}$$

Transport Equation

An important feature of Fluent is the User Defined Scalar (UDS) transport equation solver. The UDS transport equation is given for a generic scalar ϕ in Eq. (6) [11]. The variable Γ is a diffusion coefficient for the scalar, and S_ϕ are source terms.

$$\rho \frac{\partial \phi}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} \phi - \Gamma \nabla \phi) = S_\phi \tag{6}$$

Equation (3), as is, cannot be sufficiently described using Eq. (6) because the transported scalar is a mixture of the variables temperature and energy. This issue can be fixed by assuming $e = c_v T$, as done by other investigators [3, 5, 6, 7]. However, the preferred solution is using a source term, which allows the direct use of energy values. The resulting governing equation as implemented into Fluent is given as

$$\nabla \cdot \left(\rho \boldsymbol{u} \frac{\partial e}{\partial T} T - k \nabla T \right) = S_S + S_L \tag{7}$$

where S_S and S_L are source terms for the sensible and latent heat, respectively.

i. Sensible Heat Source Term

The source term for the sensible heat S_S is expanded to

$$S_S = A + BT \tag{8}$$

$$A = -\left[\rho \frac{\partial e}{\partial t} \right]^* + \left[\rho \frac{\partial}{\partial t} \frac{\partial e}{\partial T} \right]^* T^* \tag{9}$$

$$B = \left[-\rho \frac{\partial}{\partial t} \frac{\partial e}{\partial T} \right]^* \tag{10}$$

The superscript $*$ denotes the values from the previous iteration. This form of the energy equation uses temperature as the transported UDS. The first term of Eq. (9) is the user specified source supplied to Fluent [10]. Additionally, the derivative of the source with respect to the UDS

is supplied to Fluent. This additional term linearizes the source term and enhance the stability of the solver. For all intents and purposes, the partial derivative $\frac{\partial e}{\partial T}$ is the specific heat of the fluid and is evaluated using a finite difference method. Fluent handles Eqs. (9) and (10) as the explicit and implicit terms of the total source term, respectively. These components are discretized in the code as Eqs. (11) and (12). The superscript 0 denotes values from the previous timestep. The subscript P denotes the value at the current cell. This form of the energy equation is stable for a continuous, positively correlated sensible heat with respect to temperature.

$$A = -\frac{\rho_P^0}{\Delta t}(e_P^* - e_P^0) + \frac{\rho_P^0}{\Delta t}\left[\frac{\partial e}{\partial T}\right]_P^0 T_P^* \tag{11}$$

$$B = -\frac{\rho_P^0}{\Delta t}\left[\frac{\partial e}{\partial T}\right]_P^0 \tag{12}$$

ii. Latent Heat Sink Term

The sink term for the latent heat S_L is given as

$$S_L = -\rho \Delta e \frac{\partial f}{\partial t} \tag{13}$$

and is discretized in the code as

$$S_L = -\frac{\rho_P^0 \Delta e^*}{\Delta t}(f_P^* - f_P^0) \tag{14}$$

No implicit terms are necessary for this sink term because the phase fraction is independent of the cell temperature. As a fluid cell undergoes evaporation phase change, the cell temperature initially rises above the saturation temperature. For condensation, this process occurs in the opposite direction. The cell energy reflects this change in temperature, and the cell is determined to have a mixed phase fraction by the EOF method. This new phase fraction is factored into the latent heat sink term which reverses the temperature change because of the

conservation of energy. Rather, when a fluid cell undergoes phase change, the EOF update algorithm shifts the influence of the sensible heat term into the latent heat term in the transport equation. The iterative nature of this procedure is visually shown on Figure 10 using reconstructed values of the conservation equation terms. These values are obtained from the first timestep of the unsteady conduction phase change verification case, which converged in 51 iterations. This method results in the cell temperature converging onto the saturation temperature.

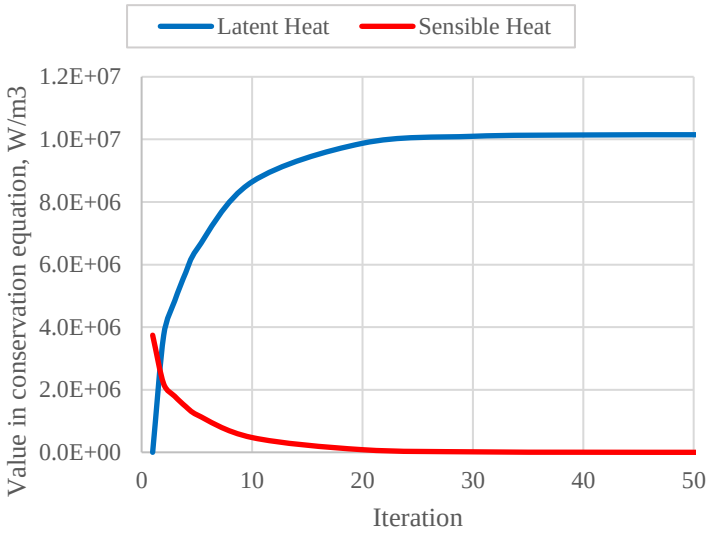


Figure 10 Effect of EOF algorithm on sensible and latent heat terms.

Special consideration needs to be taken for cells transferring heat near the phase interface. As stated, the initial temperature solution of a liquid cell undergoing evaporation is higher than the saturation temperature. The material property tables for a liquid has an upper temperature limit, the critical temperature. When the amount of heat transfer into a liquid cell raises the temperature to above the critical value, the event causes an error output from the property source code. This event can be considered flash evaporation due to the nature of the rapid phase change and must be avoided to maintain numerical stability. Sections of the outer walls in both validation cases are considered adiabatic to help prevent this issue.

iii. Discretized Transport Equation

The spatially and temporally discretized governing equation with source terms is expressed using the general linearized transport equation [15]. In Eq. (17), the diffusion and convection terms are discretized by Fluent.

$$a_P T_P = \sum a_{nb} T_{nb} + b \tag{15}$$

$$a_P = \frac{\rho_P^0}{\Delta t}\left[\frac{\partial e}{\partial T}\right]_P^0 + \sum a_{nb} \tag{16}$$

$$a_{nb} = D_{face} + F_{face}\left(\frac{\partial e}{\partial T}\right)_{face}^{*} \tag{17}$$

$$b = -\frac{\rho_P^0}{\Delta t}(e_P^* - e_P^0) + \frac{\rho_P^0}{\Delta t}\left[\frac{\partial e}{\partial T}\right]_P^0 T_P^* - \frac{\rho_P^0 \Delta e^*}{\Delta t}(f_P^* - f_P^0) \tag{18}$$

Convection Considerations

i. Buoyancy Source Term

Natural convection is modeled by adding the buoyancy source term, Eq. (19), into the momentum equations. The gravity g is dependent upon the case. The variable ρ is the cell density, which is dependent on the cell temperature and saturation pressure. The variable ρ_{ref} is a reference density. The reference density is calculated as the minimum phase-average density, which accurately reconstructs the static pressure gradient within the fluid domain [10]. For the two validation cases, the reference density is the volume average vapor density calculated explicitly from the vapor cells.

$$S = g(\rho - \rho_{\text{ref}}) \tag{19}$$

ii. Velocity Suppression Method

Addition of convection increases the complexity of the problem significantly. The major problem that occurs with convection is keeping the two fluids immiscible. That is, momentum from the liquid region must not penetrate the interface into the vapor region and vice versa. The solution presented in this work prohibits flow into and out of the interface region using a momentum sink term. This solution is inspired by a method to simulate inactive cells presented by Patankar [15]. The conservation equation for the general quantity ϕ is used to illustrate the velocity suppression method. When the source terms are much larger in magnitude than all other terms, the source-dominant conservation equation is reduced and rearranged to become Eq. (20).

$$\phi_P = \phi_P^* - \frac{S^*}{\left(\frac{\partial S}{\partial \phi}\right)^*} \tag{20}$$

The goal is to reduce ϕ_P to zero. Thus, the explicit portion of the source term is

$$S^* = \left(\frac{\partial S}{\partial \phi}\right)^* \phi_P^* \tag{21}$$

The source magnitude $\left(\frac{\partial S}{\partial \phi}\right)^*$ must be significantly larger than the rest of the conservation equation for this method to function properly and suppress the velocity. This method creates a massive pressure gradient that simulates an impermeable wall. When creating such a pressure gradient in the flow field, divergence in the pressure correction equation is likely to happen. Therefore, the source term must not create an unresolvable pressure gradient when activating and deactivating.

To implement this velocity suppression method and satisfy the pressure gradient, an algorithm is added to the UDF to mark mixed phase cells for the source term. The algorithm is executed at the end of every timestep to avoid complications with the iterative flow field calculation. At initialization, marked cells start with the highest magnitude. As a cell undergoes phase change, it is marked by the algorithm. Each timestep, the magnitude of the source term for that cell gradually increases to the highest magnitude to not create an unresolvable pressure gradient. As the magnitude increases, the cell begins to act like a porous region until the porosity become high enough to completely stop all fluid motion in the cell. When a mixed phase cell becomes pure phase, the algorithm gradually decreases the magnitude of the source term until it becomes insignificant to the momentum equation.

iii. Interface Reconstruction Method

An additional solution is implemented to aid the velocity suppression method in keeping the two phases immiscible. On occasion, heat transferred to the interface is unbalanced between the inner and outer most regions of the interface. This occurrence causes the interface to become nonphysical when gravity is present.

The Bond number is used to show the ratio between gravitational and capillary forces at the phase interface [20]. The Bond number can also be used to predict the shape of the interface. Table 3 shows the Bond number analysis for both validation cases at the initial conditions. The high Bond number magnitudes ($\mathrm{Bo} > 100$) indicate that the interface is largely unaffected by surface tension and, consequently, flat [20, 21, 22].

Table 3 Bond number analysis for validation cases.

Case	Phase Density Difference	Gravitational Acceleration	Tank Radius	Surface Tension	Bond Number
	$\Delta\boldsymbol{\rho} = \boldsymbol{\rho_l} - \boldsymbol{\rho_g}$, kg/m^3	$\boldsymbol{g}$, m/s^2	$\boldsymbol{R}$, m	$\boldsymbol{\sigma}$, N/m	$\mathrm{Bo} = \frac{\Delta\boldsymbol{\rho} \boldsymbol{g} \boldsymbol{R^2}}{\boldsymbol{\sigma}}$
Low Gravity	70.274	3.43E-3	3.3	2.01543E-3	1.30E3
Normal Gravity	69.457	9.81	0.115	1.92456E-3	4.68E3

Using the Bond number analysis, the shape of the interface is modeled to be flat, i.e. a contact angle of 90°. The phase fraction values of the cells encompassing the interface line reflect the location of the interface. At initialization, the location of the interface determines the phase fraction values. During the simulation, the phase fraction values will determine the location of the interface. Each timestep, the code uses the converged solution to find the liquid content within the tank and calculates the location of the interface using a root-finding method. Afterward, the code reassigns phase fraction values according to the new interface location.

iv. Phase Convection Flux Term

The phase fraction flux term $\Delta e \nabla \cdot (\rho \boldsymbol{u} f)$ is absent from Eq. (7) because the implementation of the term into the UDF is likely too complex. Fluent requires the convection flux term of the transport equation to be a quantity to be multiplied by temperature, i.e., if the convective flux term is $\rho \boldsymbol{u} c_v T$, then the UDF must specify $\rho \boldsymbol{u} c_v$ [10]. Implementing the phase fraction flux term would be particularly unconventional because the term will need to be specified as $\rho \boldsymbol{u} f \Delta e / T$. An alternative solution would be to implement the phase fraction flux as a source term. However, this method substantially increases complexity of the code. Therefore, the phase fraction flux term is not included in the UDS transport equation.

Solution Procedure

i. Initialization

The user function DEFINE_INIT is used to initialize all variables. This includes patching the cell temperatures, all UDM variables, and the initial saturation properties. The phase fractions at the interface are calculated using a linear interpolation scheme shown in Eq. (22). The scheme is based on the initial interface location and the coordinates of the cell node extremes. It assumes that the phase interface is flat and perpendicular to the x-axis. The location of the interface may be adjusted to match the initial fill level or mass within the tank.

$$f_{\text{initial}} = \begin{cases} 0 & x_{\text{interface}} < x_{\text{min}} \\ \dfrac{x_{\text{interface}} - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} & x_{\text{min}} < x_{\text{interface}} < x_{\text{max}} \\ 1 & x_{\text{max}} < x_{\text{interface}} \end{cases} \tag{22}$$

During initialization, the total system mass and liquid mass are calculated using a summation of the individual cell masses, shown in Eqs. (23) and (24). The total system mass is assumed to remain constant, and any change in the liquid mass is assumed to be exchanged with the vapor mass by evaporation or condensation. The vapor mass is calculated because of this assumption in Eq. (25). This assumption ensures solution stability by avoiding a feedback effect between the saturation pressure and vapor density.

$$m_{\text{total}} = \sum [\rho V]_{\text{cell}} = \text{constant} \tag{23}$$

$$m_l = \sum [\rho_l V(1-f)]_{\text{cell}} \tag{24}$$

$$m_g = m_{\text{total}} - m_l \tag{25}$$

ii. Property Updates

During the simulation, the user function DEFINE_ADJUST is used to update flow variables of the EOF algorithm. These variables include the cell energies, saturation properties,

material properties. This function is executed before Fluent solves any governing equations, including the energy equation. The following procedure is developed to implement the EOF method and update the solution variables.

With a temperature solution, the bulk vapor temperature may be calculated using a volume-weighted average over the entire domain. The bulk vapor density is calculated using the vapor mass obtained by Eq. (25).

$$T_{\text{bulk}} = \frac{\sum[TVf]_{\text{cell}}}{\sum[Vf]_{\text{cell}}} \tag{26}$$

$$\rho_{\text{bulk}} = \frac{m_g}{\sum[Vf]_{\text{cell}}} \tag{27}$$

The saturation pressure is a property that is constant throughout the entire domain. Unless stated otherwise, all properties are calculated using the methods discussed in the Material Properties section. The saturation pressure is calculated using the bulk vapor temperature and density. The corresponding saturation temperature is calculated using the new saturation pressure. The cell internal energies are determined using the new values for cell temperature and saturation pressure. The latent heat is found by subtracting the internal energies of a saturated vapor and a saturated liquid. The cell phase fractions are updated with the new cell energies using Eq. (4). From Eq. (2), the sensible heat can be found by subtracting the total internal energy by the latent heat content. The remaining material properties, such as the density, specific heat, thermal conductivity, and viscosity, are updated using the new values for the phase fraction, temperature, and pressure.

The evaporated liquid fraction (ELF) is a measure of the change in liquid mass from initialization and is calculated using Eq. (28). This measurement is used for the purposes of reporting a boil-off value and judging timestep convergence.

$$\text{ELF} = 1 - \frac{m_l}{m_{l,initial}} \tag{28}$$

iii. Judging Convergence

The residuals for the saturation pressure, ELF, and phase fraction are calculated using Eq. (29), a generic error formula for a variable ϕ. The residuals for saturation pressure and ELF directly use their respective values for ϕ. The residual for the phase fraction is calculated using a slightly different method. Originally, the maximum change in phase fraction of a cell is used to judge convergence. However, this form often leads to an oscillating convergence that may take an unnecessary number of iterations to resolve. A more tolerant form to judge phase fraction convergence uses the change in the sum of mixed phase fraction values.

$$R_\phi = \left| \frac{\phi^* - \phi}{\phi^*} \right| \tag{29}$$

Additionally, timestep convergence is judged using solver residuals provided by Fluent for the continuity, momentum, and UDS energy equations. The tolerances enforced for convergence are shown on Table 4. Once timestep convergence is achieved, the function EXECUTE_AT_END is called to store the current timestep UDM variables into previous timestep versions for use in the next timestep. This function also executes the velocity suppression and interface reconstruction algorithms.

Table 4 Timestep convergence criteria for solution residuals.

Continuity	Momentum	UDS Energy	Phase Fraction	Saturation Pressure	ELF
1E-4	1E-6	1E-6	1E-6	1E-6	1E-6

Due to the nature of Fluent, timestep convergence cannot be called using both solver residuals and the additional user defined convergence criteria. Fluent will call convergence when

either are below the specified tolerances, whichever comes first. A solution for this issue is the use of a special report definition aptly called, *timestep-ender*. Convergence is judged in the source code at the end of the DEFINE_ADJUST function. Fluent formally recognizes this judgement at the end of the iteration loop using the convergence criteria feature within the residual monitor. This method means that Fluent solves the governing equations one more time after convergence is called. However, this is negligible when the solution residuals approach zero.

Oscillations occur during the simulation, which lead to non-convergence. Therefore, relaxation factors are utilized in the calculation of the flow variables to ease the solution to convergence. The general relaxation equation is given by

$$\phi = \chi\phi + (1 - \chi)\phi^{*} \tag{30}$$

where χ is the relaxation factor. The residuals for the saturation pressure and the individual cell phase fractions are calculated before relaxation to avoid underestimated residuals. Table 5 shows the relaxation factors used for each of the flow variables.

Table 5 Relaxation factors for flow variables.

Pressure	Momentum	UDS Energy	Saturation Pressure	Phase Fraction
0.1	0.1	0.1	0.3	0.7

CODE VERIFICATION

Code verification studies confirm that the numerical solutions obtained with the EOF method are accurate with published analytical solutions. Water is used as the heat transfer medium for all verification cases in this research. The results of the code verification cases show that the simulation predictions agree with analytical predictions.

Unsteady Conduction Heat Transfer

The governing energy equation for a one-dimensional case with only unsteady conduction is simplified to yield

$$\rho c_v \frac{\partial T}{\partial t} = k \frac{\partial^2 T}{\partial y^2} \tag{31}$$

Two cases verify the implementation of the governing equation in this simplified form. For these cases, the specific heat is a known value, and calculation of the internal energy is not necessary. Therefore, temperature is the transported scalar in the energy equation.

For both cases, the same 0.1 m x 0.1 m two-dimensional square geometry is used in a uniform 100x100 cell computational mesh. The simulations are run to a flow time of 600 sec at a time step size of 1 sec. Each time step is iterated until the solution residuals are converged according to the previously stated criteria. The only solution residual monitored in these cases is the energy equation. Table 6 details the constant material properties of water used in the simulations, which are at a temperature of 300 K. The saturation temperature of water is 373.15 K at a pressure of 101,325 Pa. No phase change occurs.

Table 6 Material properties of liquid water at 101,325 Pa and 300 K.

Thermal Conductivity	**Density**	**Specific Heat**	**Thermal Diffusivity**
k**, W/m-K**	ρ**, kg/m^3**	c_v**, J/kg-K**	α**, m^2/s**
0.6102	996.5298	4130.0	1.4646E-7

i. Temperature Boundary Condition

Equation (32) provides an analytical solution for the temperature at a distance y at time t for a region of $0 < y < L$, where the temperature is T_0 at $y = 0$ and T_L at $y = L$ [16].

$$T(y,t) = T_0 - (T_0 - T_L)\frac{y}{L} - \frac{2}{\pi}(T_0 - T_L)\sum_{z=1}^{\infty}\left[\frac{1}{z}\sin\left(z\pi\frac{y}{L}\right)\exp\left(-z^2\pi^2\frac{\alpha t}{L^2}\right)\right] \quad (32)$$

The boundary conditions are detailed in Figure 11. Along the top wall, the temperature boundary condition is 300 K. At the bottom wall, the temperature boundary condition is 325 K. The two side walls are adiabatic. The initial temperature of the fluid is 300 K. Figure 12 shows the results of the simulation at 600 sec, which are in good agreement with the analytical solution.

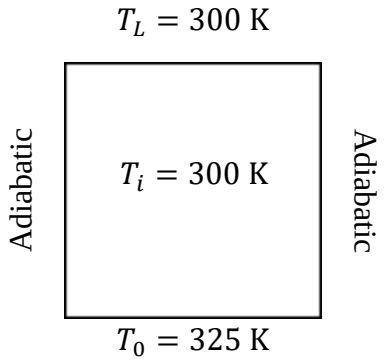


Figure 11 Unsteady conduction heat transfer: Temperature boundary conditions and initial conditions.

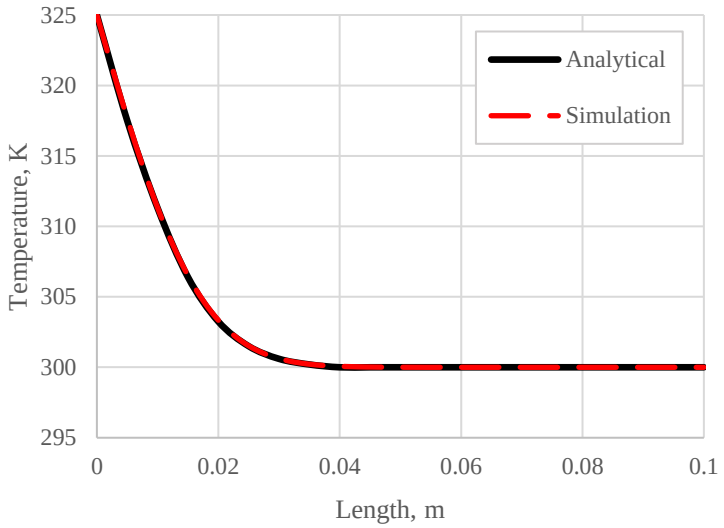


Figure 12 Unsteady conduction heat transfer: Comparison of numerical and analytical results of temperature boundary condition case.

ii. Heat Flux Boundary Condition

Equation (33) provides an analytical solution for a similar scenario as the previous case, where the heat flux q is present at $y = L$ [17].

$$T(y,t) = T_i + \frac{2q(\alpha t)^{1/2}}{k}\sum_{z=0}^{\infty}\left[\text{ierfc}\frac{(2z+1)(L-y)}{2(\alpha t)^{1/2}} + \text{ierfc}\frac{(2z+1)(L+y)}{2(\alpha t)^{1/2}}\right] \tag{33}$$

$$\text{ierfc}(x) = \frac{1}{\sqrt{\pi}}e^{-x^2} - x[\text{ierfc}(x)] \tag{34}$$

The boundary conditions are detailed in Figure 13. At the top wall, the heat flux boundary condition is 200 W/m^2. All other walls are adiabatic. The fluid is initialized at 300 K. Figure 14 shows the results from the simulation at 600 sec, which are in good agreement with the analytical solution.

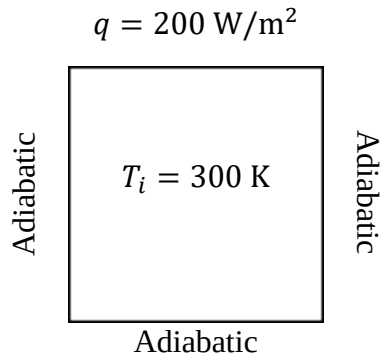


Figure 13 Unsteady conduction heat transfer: Heat flux boundary conditions and temperature initial conditions.

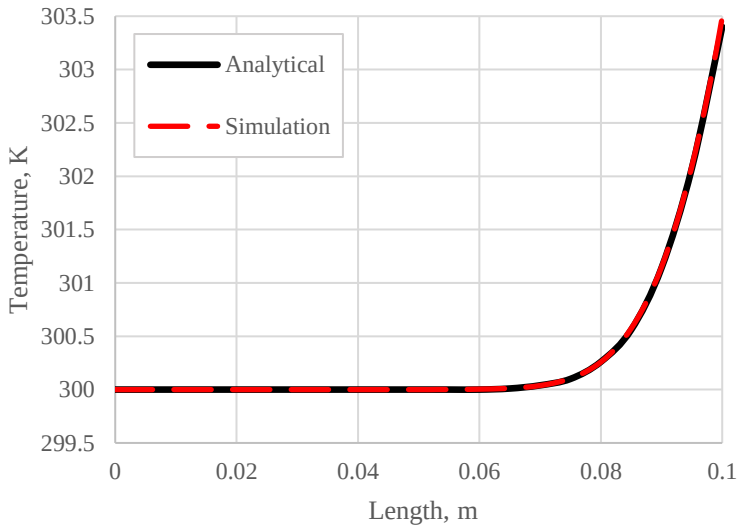


Figure 14 Unsteady conduction heat transfer: Comparison of numerical and analytical results of heat flux boundary condition case.

Unsteady Conduction Phase Change

The case described in this part of the section is used to verify the phase change implementation of the model. The governing equation in this case is

$$\rho c_v \frac{\partial T}{\partial t} = k \frac{\partial^2 T}{\partial y^2} - \rho \Delta e \frac{\partial f}{\partial t} \tag{35}$$

Phase change in this case is one of solidification and melting and occurs purely due to conduction heat transfer. An analytical solution for this one-dimensional solidification case is given as

$$Y = 2\lambda(\alpha t)^{1/2} \tag{36}$$

where $\lambda = 0.79$ and α is of the solid phase [17]. Table 7 shows the relevant material properties of the fluid. The freezing point is set at 275.15 K with a latent heat value of 334,944 J/kg.

Table 7 Material properties of liquid and solid water at 101,325 Pa.

Phase	**Thermal Conductivity**	**Density**	**Specific Heat**	**Thermal Diffusivity**
	k**, W/m-K**	ρ**, kg/m^3**	c_v**, J/kg-K**	α**, m^2/s**
Liquid	0.6209	1000.0	4186.8	1.4400E-7
Solid	2.2190	920.0	2101.8	1.1476E-6

The boundary conditions are detailed in Figure 15. The top wall temperature is 275.15 K. The bottom wall temperature is 273.15 K, which is where solidification originates. The side walls are adiabatic. The initial fluid temperature is 275.15 K. The simulations are run to a flow time of 600 sec using a time step size of 1 sec. The solution residuals monitored in this case are the energy equation and the phase fraction.

Smearing the phase interface across cells suggests that the solution has an inherent mesh dependency that must be resolved. However, the length scale at which phase change occurs is along the molecular scale. Therefore, mesh convergence studies are judged on a relative error

basis. A mesh convergence study is conducted using a 0.0025 m x 0.1 m two-dimensional geometry of varying mesh sizes. The results of study are shown on Figure 16 alongside the results from the analytical solution. The legend denotes grid size. As the mesh is refined, the numerical solution nearly converges onto the analytical solution. The results are shown to be in good agreement.

Location of the phase interface is calculated with an interpolation scheme using the cell geometry and cell-centered phase fraction. The scheme is given by

$$Y = y_{\text{cell}} + (0.5 - f)\Delta y \tag{37}$$

where y_{cell} is the y-coordinate of the cell centroid and Δy is the vertical length of the uniform square cell.

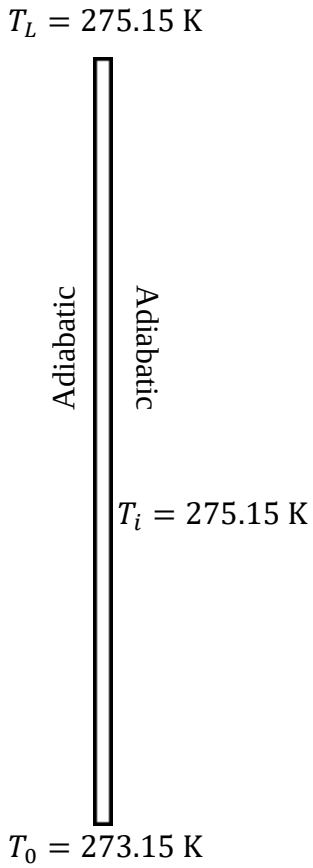


Figure 15 Unsteady conduction phase change: Temperature boundary conditions and initial conditions.

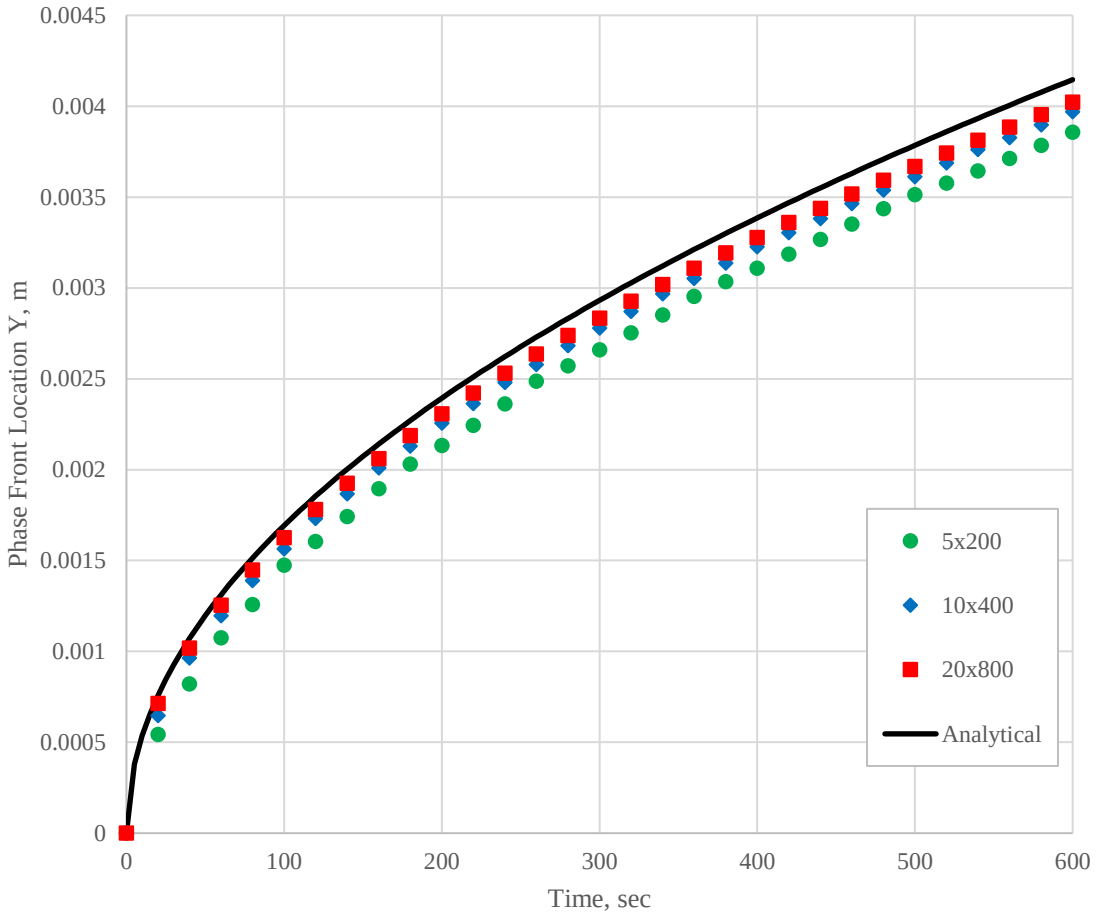


Figure 16 Unsteady conduction phase change: Comparison of analytical and mesh dependent numerical results using phase front location.

CODE VALIDATION

Initial Model

i. Recreation of Results

This work builds upon the conduction model developed by Winter. An initial source code is developed to recreate the results obtained by Winter. This code uses the current UDF implementation into Fluent as described in the Computational Methods section. Convection and all consequential features, such as the interface reconstruction, are absent in this case. The geometry used by Winter and for the recreation herein is a three-dimensional 15-degree wedge slice of the fuel tank. The mesh consists of 6951 tetrahedral cells generated with a 13-inch global seed size. The heat flux boundary conditions used are the heating rates determined by Bradshaw. Certain portions of the boundary hull are modeled as adiabatic to avoid flash evaporation and solution divergence [3].

A comparison between the original Winter results and the recreated simulation is shown on Figure 17. The discrepancy between the two models is likely due to a difference discretization of the energy equation, specifically the thermal conductivity in the influence coefficients. The Winter model discretizes the energy equation within the UDF itself and uses a harmonic mean of the current and neighboring cell. The energy equation used to recreate these results is discretized by Fluent using a First Order Upwind method; the thermal conductivity in the influence coefficient is that of the neighboring cell [15].

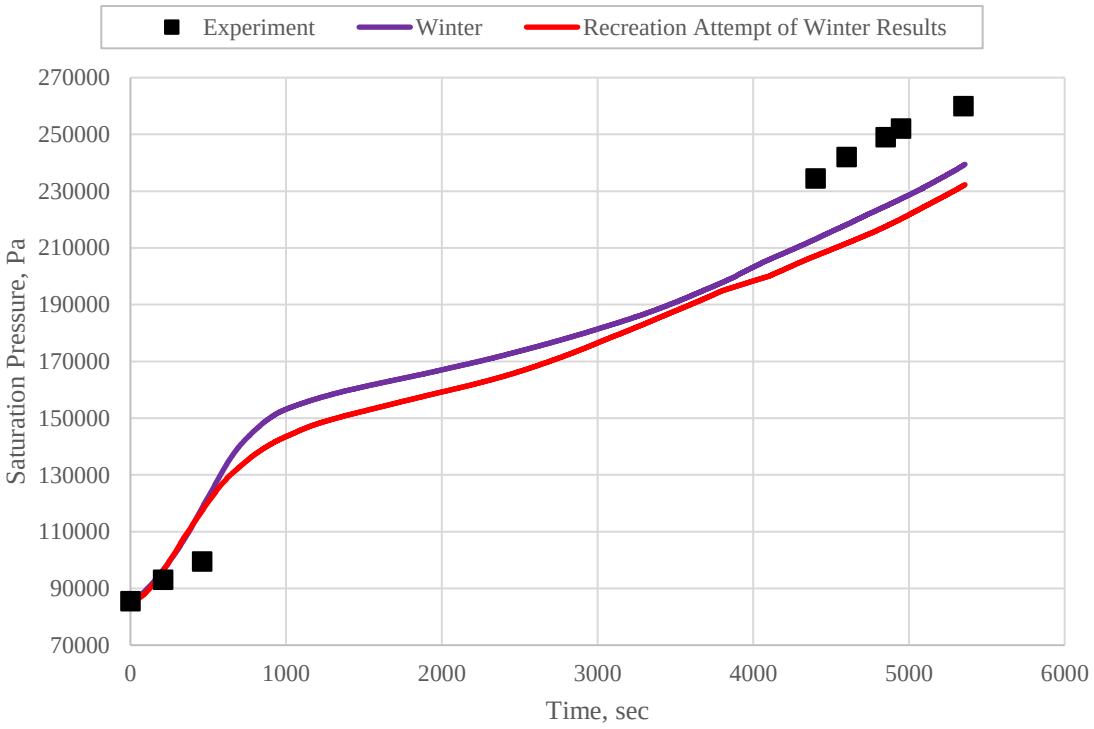
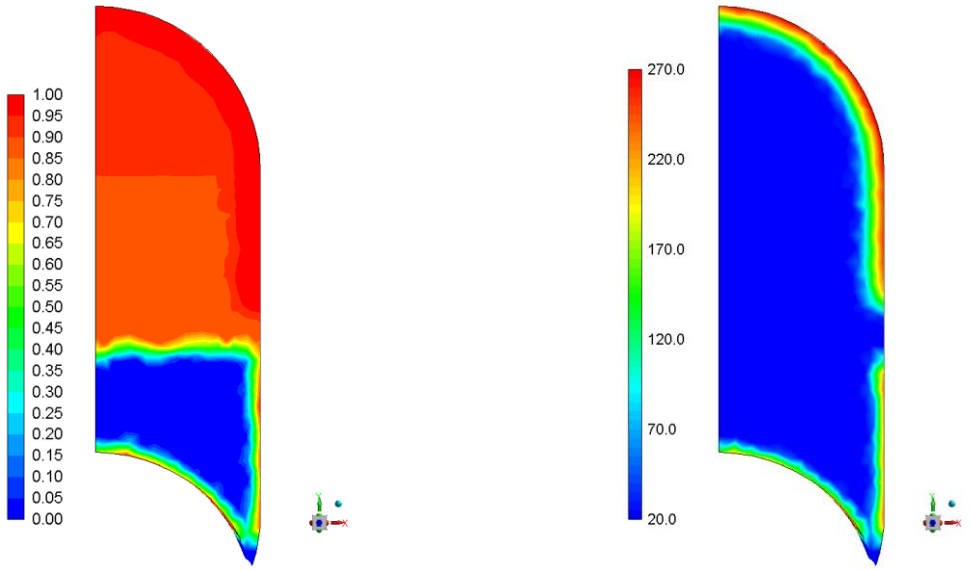


Figure 17 Initial model: Pressure rise comparison.



(a) **(b)**

Figure 18 Initial model: Final distributions for (a) phase fractions and (b) temperatures.

ii. Improvements to Initial Model

Although the initial model can adequately predict the pressure rise, several improvements were implemented to satisfy other forms of validation criteria. For example, the governing equation in this initial UDF assumed $e = c_v T$, which did not account for the refence temperature. Usually, this is negligible when the energy is directly obtained by this assumption. However, the model referred to tabulated values for the internal energy. As a first solution, the governing equation utilized the energy tables in conjunction with the preconditioned source term described in the Computational Methods section. This solution was later improved significantly by overhauling the calculation methods for the material properties.

All material properties in the initial model were obtained by linear interpolation from a series of data tables from the National Institute of Standards and Technology (NIST) Chemistry Webbook [11]. After an extensive number of simulations, a more comprehensive thermodynamic model of the hydrogen properties was necessary to accurately model the fluid under a wide range of temperatures and pressures. An option was considered to supplement the existing data tables with an abundant number of data points to fill in the curvature, but this option was not chosen since it was not guaranteed that the curvature would be resolved. Curve fits for the equation of state and derived properties were the preferred choice because they include the nonlinearities that an interpolation scheme would omit between data points.

The boundary conditions applied to the tank walls are also adjusted. Originally, the heat flux applied to the liquid region is present in the common bulkhead. This boundary condition is non-physical because on the other side of the common bulkhead is the ullage space for the LOX tank. Ward presents heat flux data separated by wall sections, and several simulations were

conducted using these boundary conditions. This heat flux data is obtained using a conduction method based on temperature differences through the walls. The temperature difference method results in much higher heat flux values than the methods of Bradshaw. Consistent with Bradshaw's conclusions, the resulting pressure rise using these boundary conditions is consequently much higher as well. Therefore, the Bradshaw heat fluxes are still the preferred choice.

Convection is added to the model using the techniques stated in the Computational Methods section. Consequently, the deflector and baffle present in the tank need to be implemented to the model. These features mitigate sloshing within the tank and affect the development of the flow field. They are assumed to not be a source of heat leak and are modeled as adiabatic line surfaces. Locations of the deflector and baffle are presented in the next section.

All following simulations use two-dimensional axisymmetric geometries and meshes. Since the energy is a scalar equation, the change in coordinate system does not affect the implementation. This change saves computational time while achieving higher fidelity meshes. With these improvements, the pressure rise within the tank can be adequately predicted along with the phase, temperature, and velocity distributions.

Low Gravity Large Scale Orbital Fuel Tank

i. Computational Setup

Figure 19 shows the two-dimensional axisymmetric geometry used in the simulations and the designations given to the wall sections. Fluent requires that the axis of symmetry coincide with the x-axis [9]. Therefore, the geometry is oriented as such.

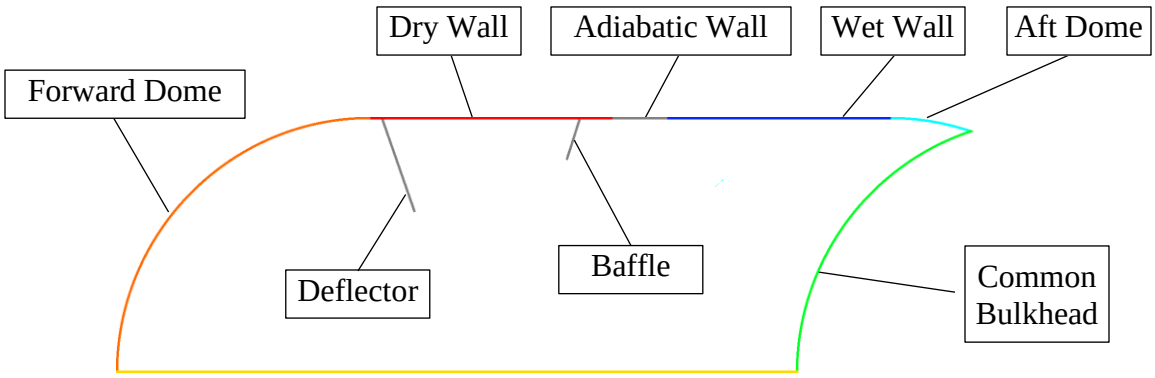


Figure 19 Low gravity setup: Wall designations.

The heat rates are extracted from data found by Bradshaw and shifted in time to start at $t = 0$. Sixth order polynomial fits of the heat rate data are shown in Figure 20 and given in Eqs. (38) and (39).

$$Q_{\text{dry}} = (-1.36820\text{E-}17)t^6 + (1.91965\text{E-}13)t^5 - (1.01026\text{E-}9)t^4 + (2.52341\text{E-}6)t^3 - (2.00413\text{E-}3)t^2 - (4.95785)t + (1.61470\text{E+}4) \quad (38)$$

$$Q_{\text{wet}} = (-6.16135\text{E-}17)t^6 + (9.11402\text{E-}13)t^5 - (5.03916\text{E-}9)t^4 + (1.27767\text{E-}5)t^3 - (1.25141\text{E-}2)t^2 - (3.76164)t + (2.14191\text{E+}4) \quad (39)$$

These heating rates are converted to heat fluxes by dividing the heat rate by the total wall area to which they are applied, given in Eqs. (40) and (41). The dry heat rate is applied to the dry wall and forward dome. The wet heat rate is applied to the wet wall and aft dome. Figure 21 shows the application of these heat flux boundary conditions.

$$q_{\text{dry}} = \frac{Q_{\text{dry}}}{A_{\text{dry wall}} + A_{\text{forward dome}}} \tag{40}$$

$$q_{\text{wet}} = \frac{Q_{\text{wet}}}{A_{\text{wet wall}} + A_{\text{aft dome}}} \tag{41}$$

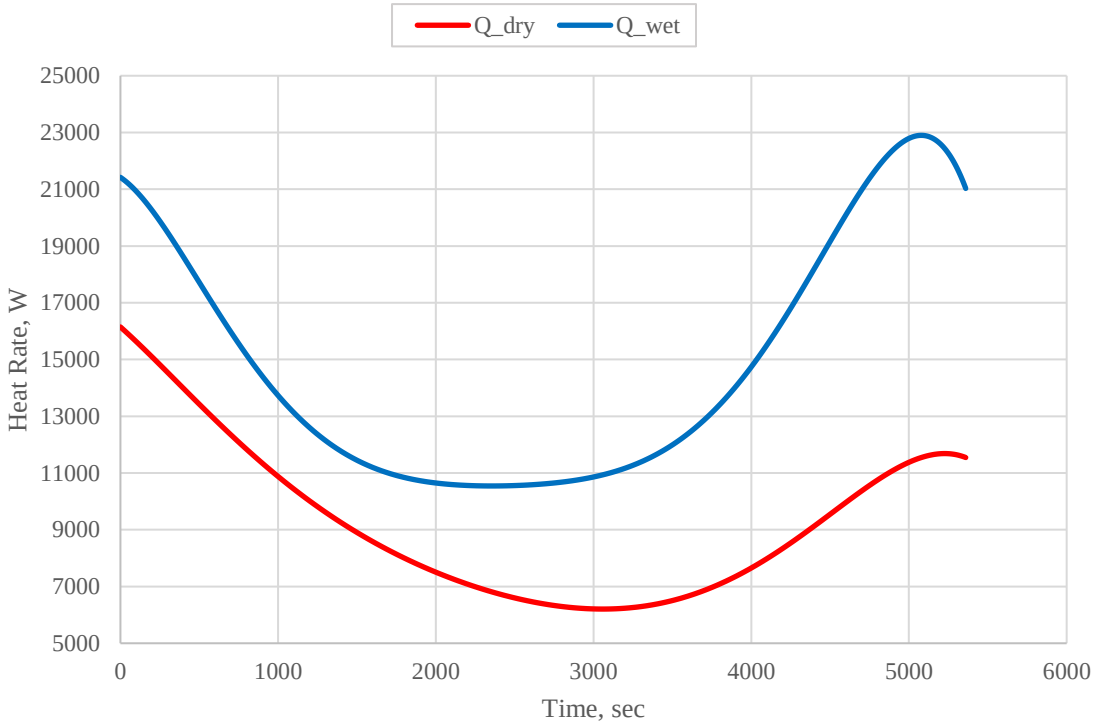


Figure 20 Low gravity setup: Polynomial fit heat rates applied to AS-203 wall boundaries.

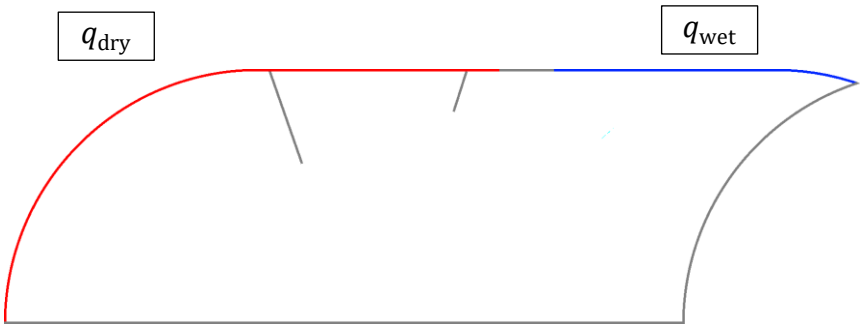


Figure 21 Low gravity setup: Applied heat flux boundary conditions.

The tank experiences gravity levels that vary with time, ranging between 3.5E-4 and 7.8E-5. This data is extracted from vehicle acceleration data from the duration of the experiment. The gravity data is fitted to the 6th order polynomial given by Eq. (42) and shown visually on Figure 22.

$$G = (3.21349\text{E-}25)t^6 - (5.81560\text{E-}21)t^5 + (4.12226\text{E-}17)t^4 - (1.45614\text{E-}13)t^3 + (2.76153\text{E-}10)t^2 - (3.17385\text{E-}7)t + (3.53157\text{E-}4) \quad (42)$$

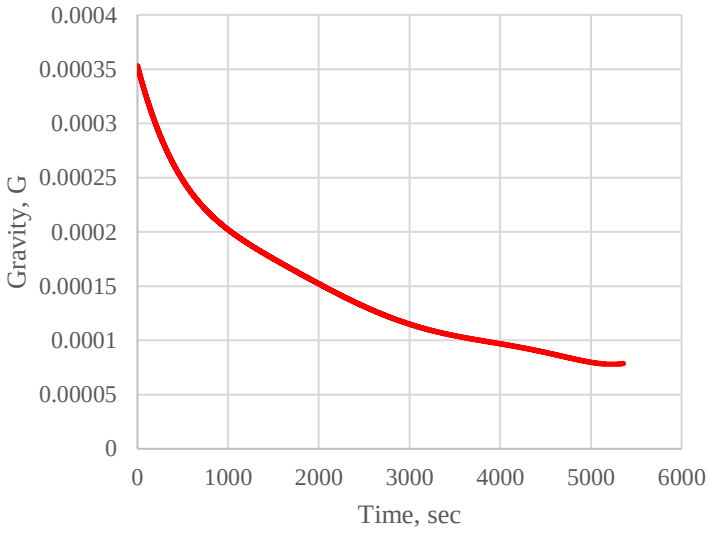
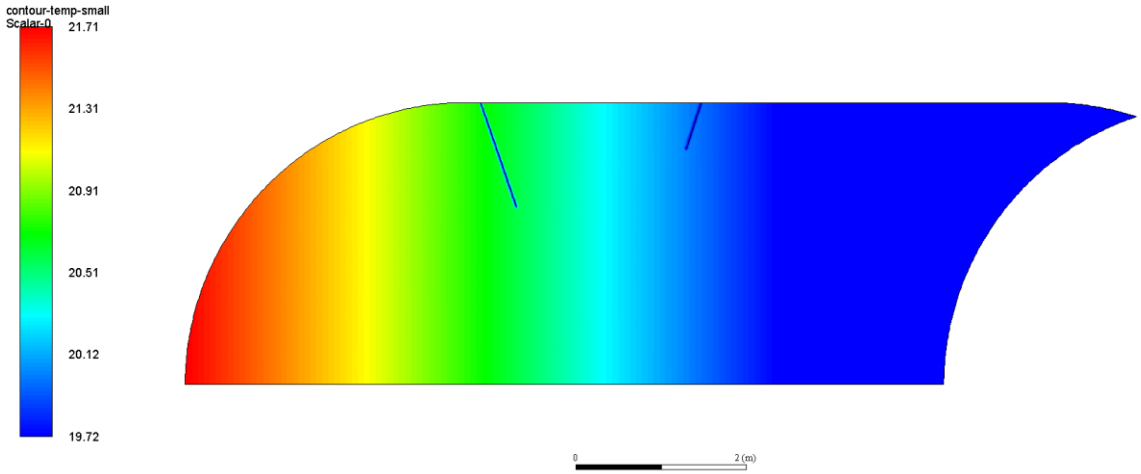
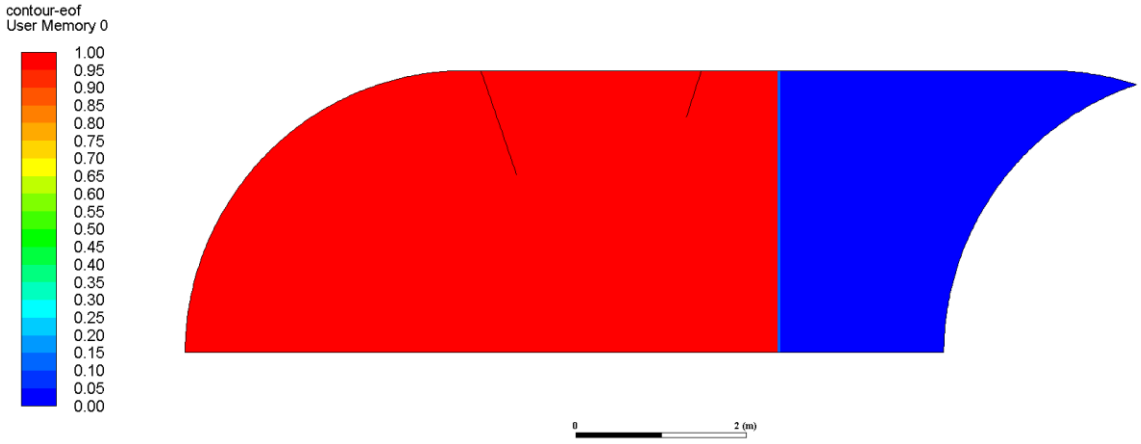


Figure 22 Low gravity setup: Transient apparent gravity data.

The initial interface location is found to be 6.961 m from the top of the tank. The interface phase fraction interpolation scheme in Eq. (22) ensures that the initial liquid mass within the tank is as close as possible to the experiment value of 7257 kg. The wall boundary adjacent to the interface is modeled as adiabatic to prevent flash evaporation. An initial temperature gradient is applied to the vapor region in accordance with data from the experiment. The rest of the fluid is initialized at the saturation temperature of 19.71 K. Figure 23 shows temperature and phase contours of the initialized domain.



(a)



(b)

Figure 23 Low gravity setup: Initial distributions for (a) temperature and (b) phase fractions.

ii. Results and Discussion

The simulations are run to a flow time of 5,360 sec at a time step size of 0.1 sec. As observed with the unsteady phase change verification case, the low gravity validation case is mesh sensitive. A mesh convergence study is conducted to show this sensitivity. The investigated mesh sizes are shown on Table 8. These meshes which consist of mostly quadrilateral cells with some triangular cells. Figure 24 shows the transient pressure data obtained with the different meshes. The mesh convergence study shows that the resulting pressure rise changes very little as the mesh is refined. The predictions diverge from each other at around 500 sec and reconverge near the end.

Table 8 Low gravity results: Mesh sizes.

Mesh Size	Cell Count
60mm	8,241
50mm	11,895
40mm	18,700
30mm	32,787

The difference between the meshes is quantified by a sum of squared residuals, Eq. (43).

$$Sum\ of\ Squares = \sum_{n=0}^{N} \left(P_i^n - P_j^n\right)^2 \tag{43}$$

The subscripts i and j denote the mesh size from which the data is retrieved. The superscript n denotes the time step, where N is the total number of time steps. Average relative errors from the four simulations are used to determine mesh convergence, shown in Table 9. According to the analysis, the error decreases when refining the mesh, suggesting that the solution is becoming more mesh independent. Due to time constraints, the extent of the mesh refinement was not investigated, and a timestep convergence study was not conducted.

Table 9 Low gravity results: Mesh convergence study.

Mesh Size Comparison	Sum of Squared Residuals
60mm – 50mm	1.55E+11
50mm – 40mm	1.15E+11
40mm – 30mm	6.97E+10

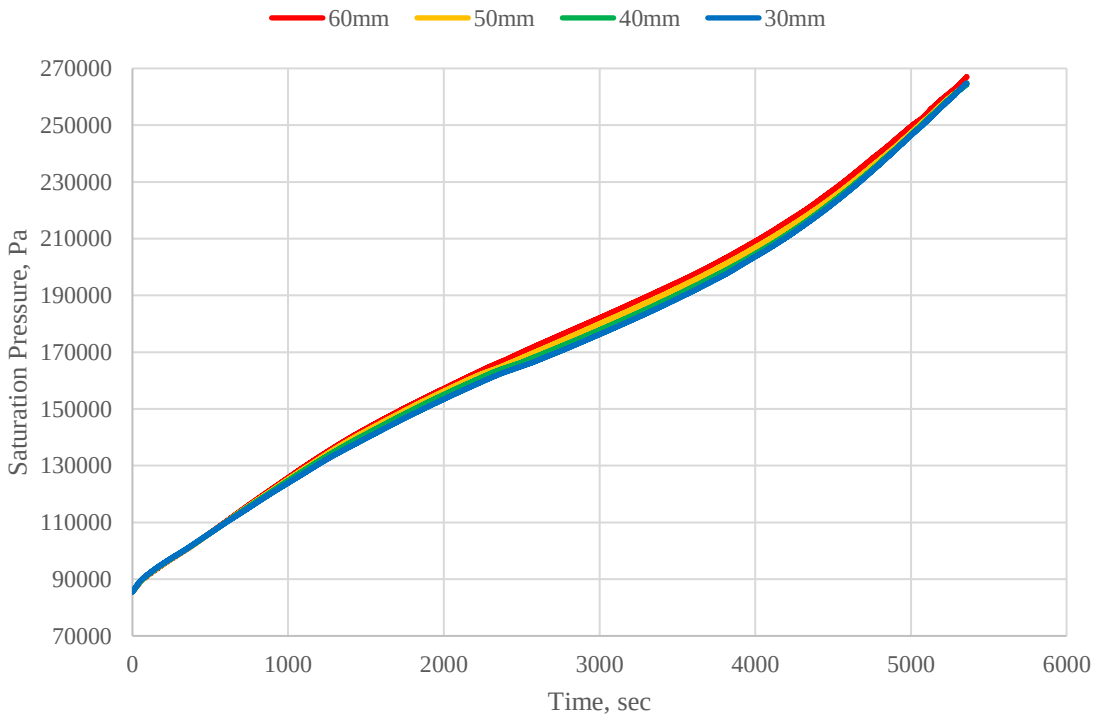


Figure 24 Low gravity results: Pressure rise at different mesh sizes.

Figure 25 shows a comparison of the results from previous works with those from the new model. All results shown hereafter are obtained using the finest mesh. The model seems to capture the pressure rise near the beginning of the simulation, which is seen in the Ward model but not in the Bradshaw or Winter models. Near the end, the model seems to be of the opposite curvature compared to the experiment and the P3542 model by Bradshaw.

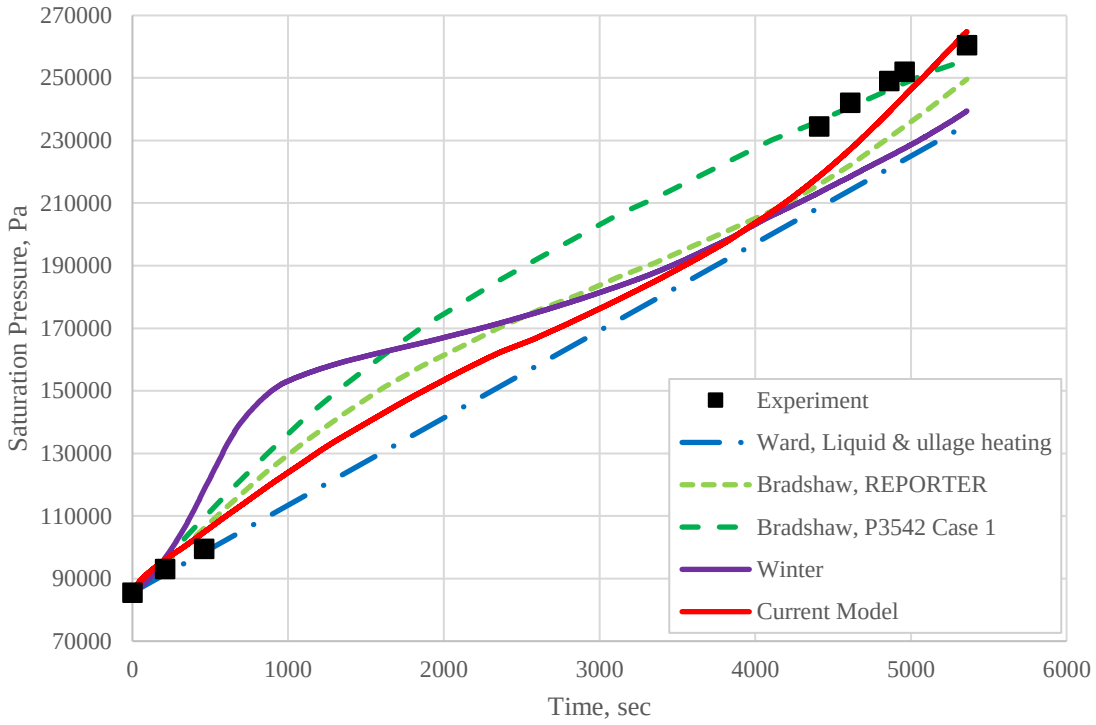


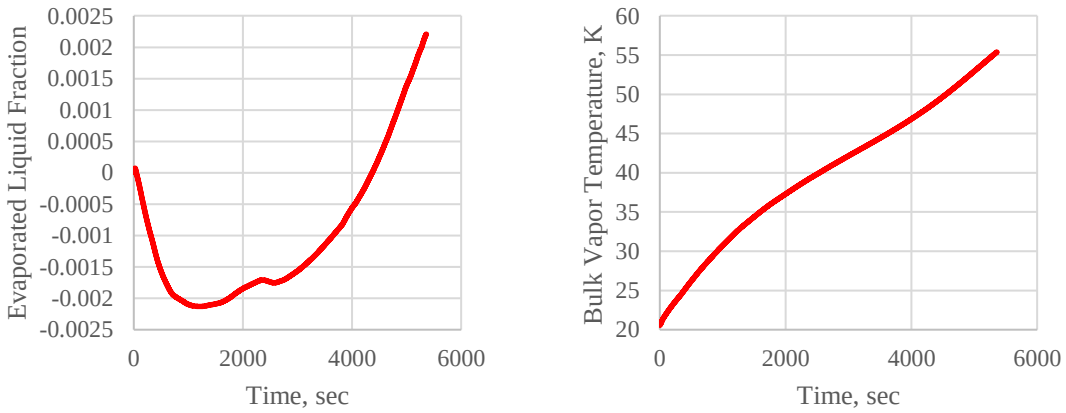
Figure 25 Low gravity results: Pressure rise comparisons with previous works.

A quantitative comparison of the experiment pressures and the current model is shown on Table 10. The largest difference found is 6.8%, while the smallest difference is 1.66%. This analysis shows that the model can adequately predict the pressure rise within the tank.

Table 10 Low gravity results: Difference of pressure prediction with experiment.

Time, sec	Experiment, Pa	Current Model, Pa	Percent Difference
210	93,000	96,013	3.24%
460	99,500	104,828	5.35%
4410	234,500	218,548	6.80%
4610	242,000	227,195	6.18%
4860	249,000	239,240	3.92%
4960	252,000	244,324	3.05%
5360	260,500	264,822	1.66%

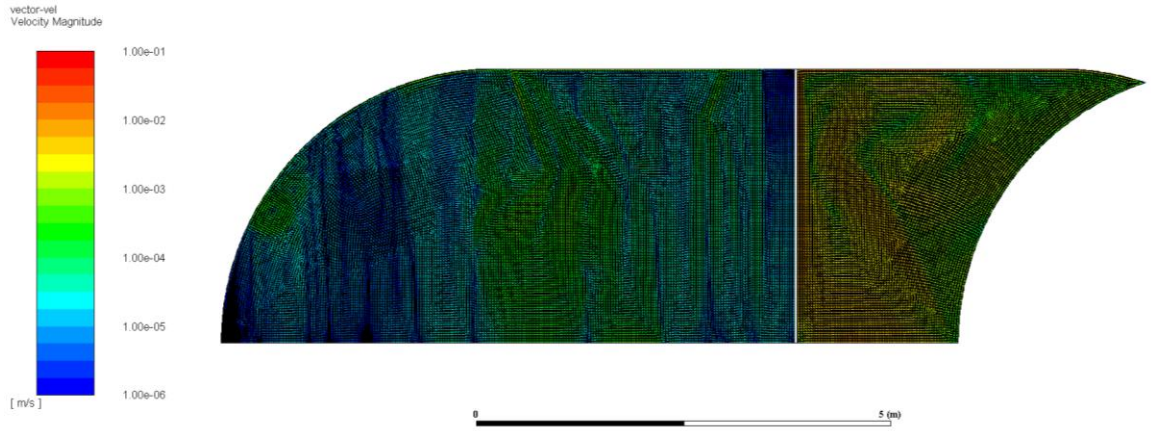
Figure 26 shows the ELF and bulk vapor temperature during the simulation. The ELF at the end of the experiment is 0.00221, or 16 kg of boiloff. This prediction is a magnitude less than the one by Ward, which is 113 kg. The actual boiloff experienced at the end of the experiment was not measured but calculated. The ELF suggests that, for most of the simulation, the tank has a net condensation until around 4,500 sec. From that point onward, the tank shows a net evaporation. As the saturation pressure rises, liquid condenses at the interface. Simultaneously, heat from the ullage evaporates the liquid at the interface. These two effects counterbalance each other for the duration of the simulation. During the beginning, the condensation rate outweighs the evaporation rate because the flow has not stratified. Heat from the ullage walls do not reach the liquid region. At around 500 sec, the ullage stratifies and allows heat to reach the interface. The evaporation rate outweighs the condensation rate, leading to a rise in the ELF.



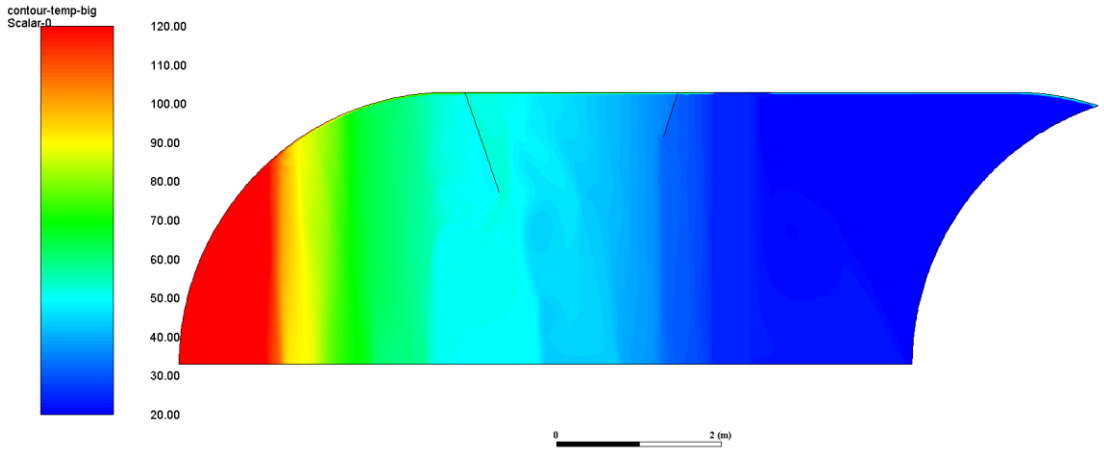
(a) **(b)**

Figure 26 Low gravity results: (a) evaporated liquid fraction and (b) bulk vapor temperature.

Figure 27 shows the velocity and temperature distributions at the end of the simulation. The ullage region shows stratification, indicated by velocity magnitudes under 1E-3. Despite having gravity levels three magnitudes lower than normal, the ullage region shows axially oriented thermal stratification, which would not be possible using a conduction model. The liquid region holds velocity magnitudes much higher than the ullage region. The resulting flow field in this region is expected considering two factors: the magnitude of the heat flow, and the low viscosity of the liquid phase.



(a)



(b)

Figure 27 Low gravity results: Final distributions at 5,360 sec for (a) velocity and (b) temperature.

Figure 28 shows the phase fraction contours at the end of the simulation. The phase distribution is vastly improved compared to that of the initial model. The addition of buoyancy driven convection allows the liquid cells near the outer wall to carry substantially more heat to the bulk liquid. Natural convection also prevents evaporation of any liquid cells near the wall.

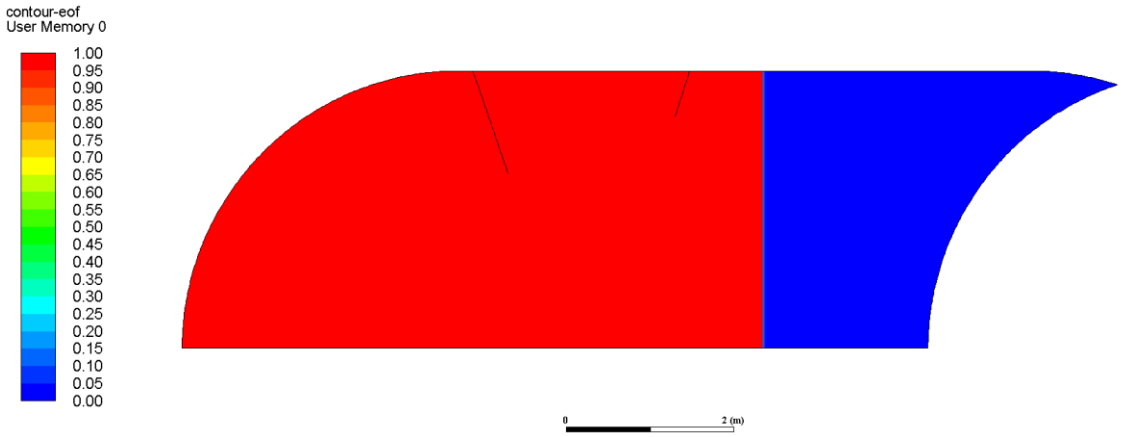


Figure 28 Low gravity results: (a) final phase distribution (b) interface location.

The location of the phase interface moves during the simulation, as shown in Figure 29. The interface location, denoted by the red line, moves from an initial axial location of 6.961 m to a final location of 6.931 m. This movement crosses cell boundaries, denoted by the black lines. The mesh is intentionally generated such that the cells near the interface are as uniform and orthogonal as possible. Consequently, the cells in this region can be categorized into columns of cells with similar phase fractions. The velocity suppression and interface reconstruction algorithms can move the simulated wall from one column of cells to another.

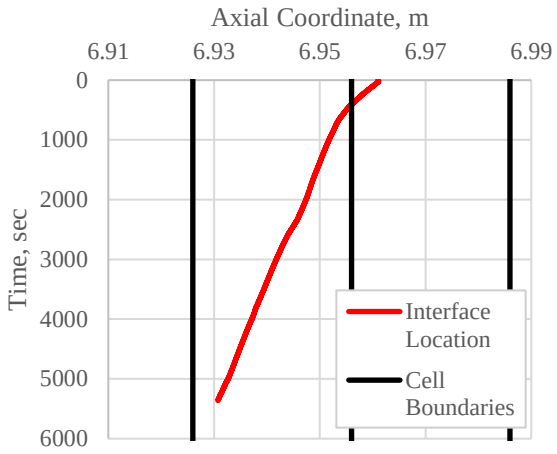


Figure 29 Low gravity results: Change in interface location.

Figure 30 presents a demonstration of the interface reconstruction using the velocity vectors captured during the simulation. The simulated interface wall is denoted by red and suppresses fluid motion. The saturation of the red corresponds to the magnitude of the source term, where fully saturated (fully red) denotes the highest magnitude. The reconstruction algorithm forms a new wall adjacent to the existing wall corresponding to the direction of phase change, which is condensation. Thus, the new wall is formed on the left. Both walls are maintained until the cells in the original wall become fully liquid. Afterwards, the algorithm gradually decreases the magnitude of the source term such that the fluid becomes unhindered and free flowing.

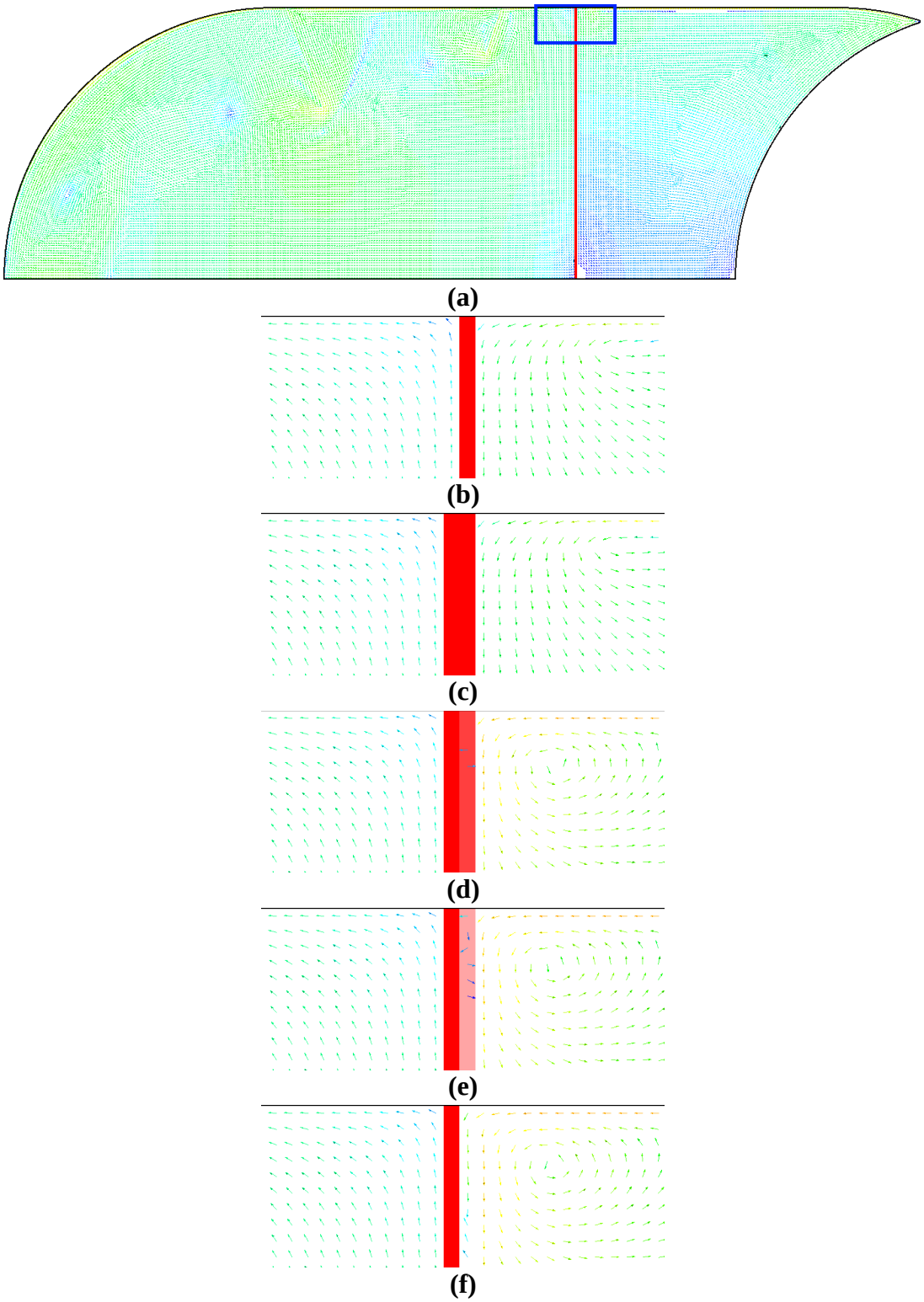


Figure 30 Low gravity results: Demonstration of interface reconstruction

(a) Vectors of velocity and location of animation frames in blue box at time =

(b) 190 sec, (c) 200 sec, (d) 410 sec, (e) 420 sec, (f) 430 sec.

Figure 31 shows a plot comparing the cell-centered temperatures in the ullage region with data provided by Ward. The plot is oriented such that the y-axis represents the height of the fuel tank with $y = 0$ being where the aft dome and common bulkhead meet. Due to the circulation currents present in the ullage, the recorded temperatures tend to fluctuate. These temperature fluctuations likely occur within the experiment but are not captured due to the low count of temperature probes in the ullage region. Hot vapor from lower parts of the tank tends to pool towards the forward dome and further heated by the heat flux boundary condition. This event causes the sharp increase in temperature near the top of the forward dome. Table 11 shows a quantitative comparison of the ullage temperatures from the experiment and the simulation. The largest difference between the results is 32.7%, while the smallest difference is 1.31%.

Table 11 Low gravity results: Comparison of ullage temperatures.

Tank Height, m	Experiment Temperature, K	Simulation Temperature, K	Percent Difference
10.75	114.4	146.8	28.3%
9.05	70.00	69.08	1.31%
7.45	44.44	49.15	10.6%
5.95	35.56	43.29	21.8%
5.39	27.78	36.87	32.7%
4.25	25.00	24.00	4.00%

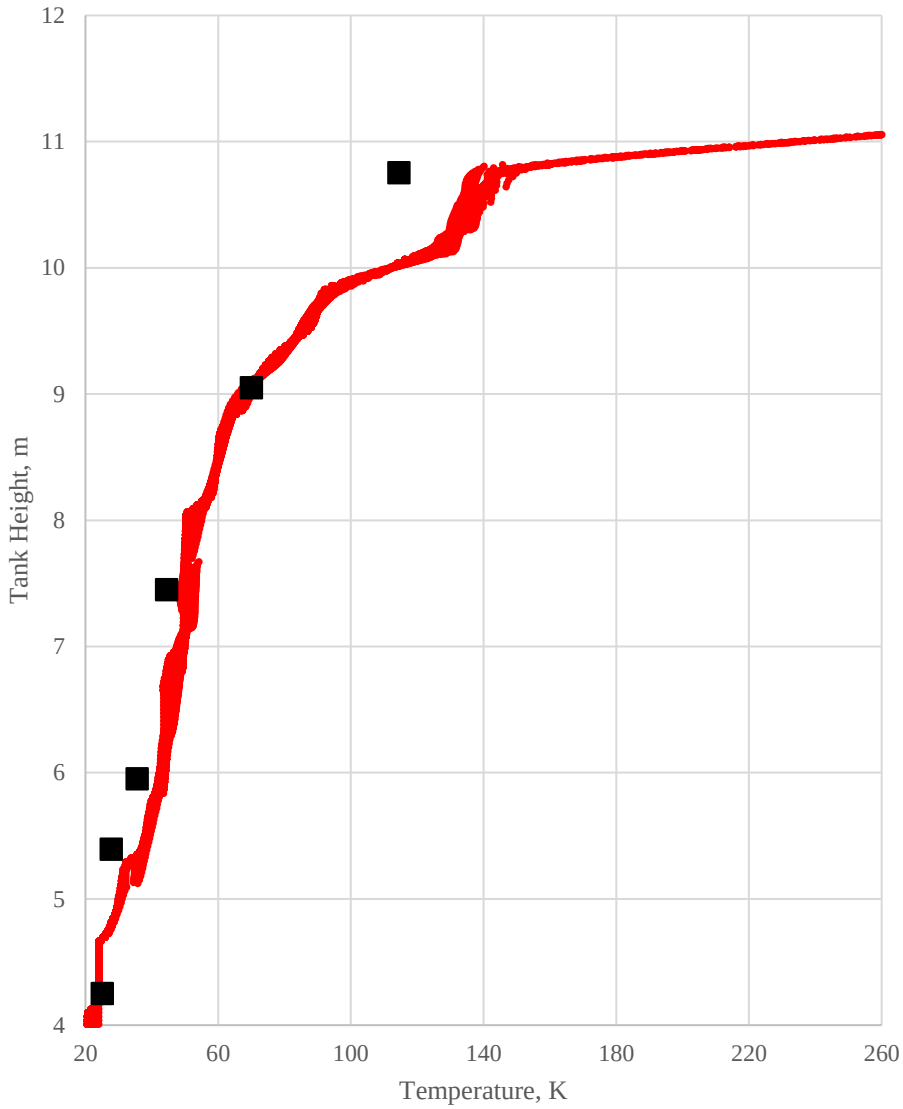


Figure 31 Low gravity results: Comparison of ullage temperature distributions.

Normal Gravity Small Scale Terrestrial Spherical Tank

i. Computational Setup

Figure 32 shows the two-dimensional axisymmetric geometry used in the simulations, the designations given to the wall sections, and the applied boundary conditions. The heat fluxes are provided by Aydelott to be 101 W/m^2 for the dry wall and 7 W/m^2 for the wet wall. Like the previous case, an adiabatic section is added to the wall near the interface to prevent flash evaporation. The initial fill level of the tank is 50.5%. The interface location that corresponds to this fill level is found to be 0.1142 m from the top of the tank. Figure 33 shows the initialized phase fractions within the fluid domain. At an initial saturation pressure of 101,325 Pa, the initial temperature of the entire domain is the corresponding saturation temperature of 20.27 K.

As an additional form of validation, the temperatures within the vapor region are recorded at certain distances from the top of the tank using line geometries created in Fluent. The temperature rakes are oriented radially and calculate an area-weighted average. The axial locations of these temperature rakes replicate those of the temperature transducers from the experiment. The specific transducer designations replicated are R-2, R-3, R-4, and R-5 [4].

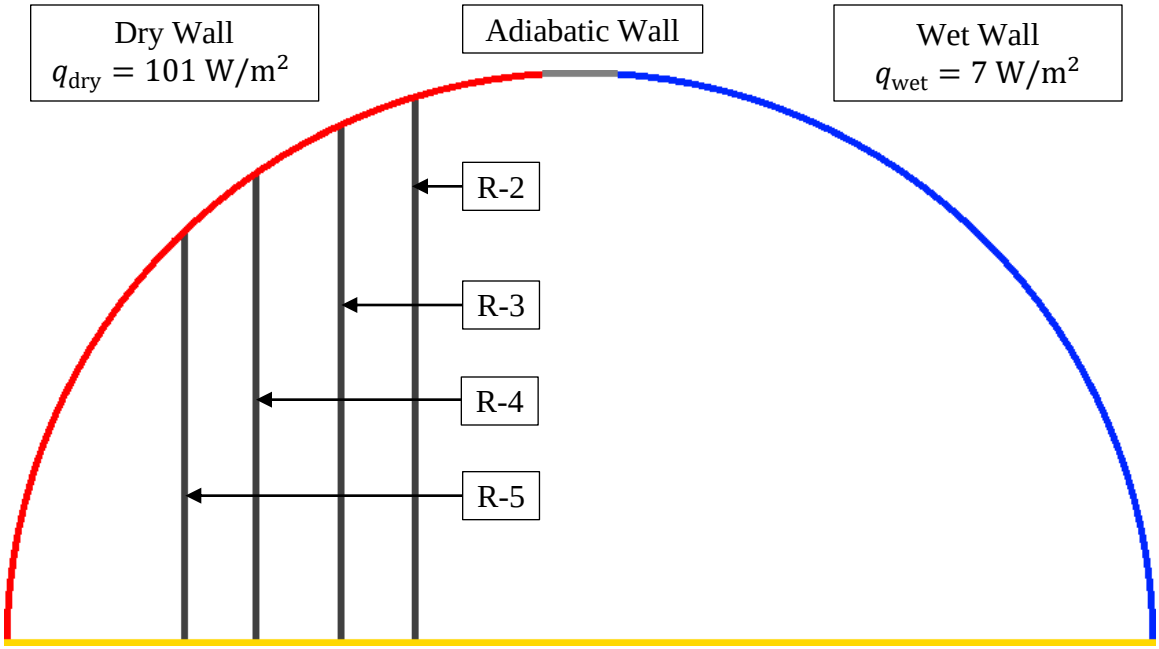


Figure 32 Normal gravity setup: Wall designations, boundary conditions, and temperature rake locations.

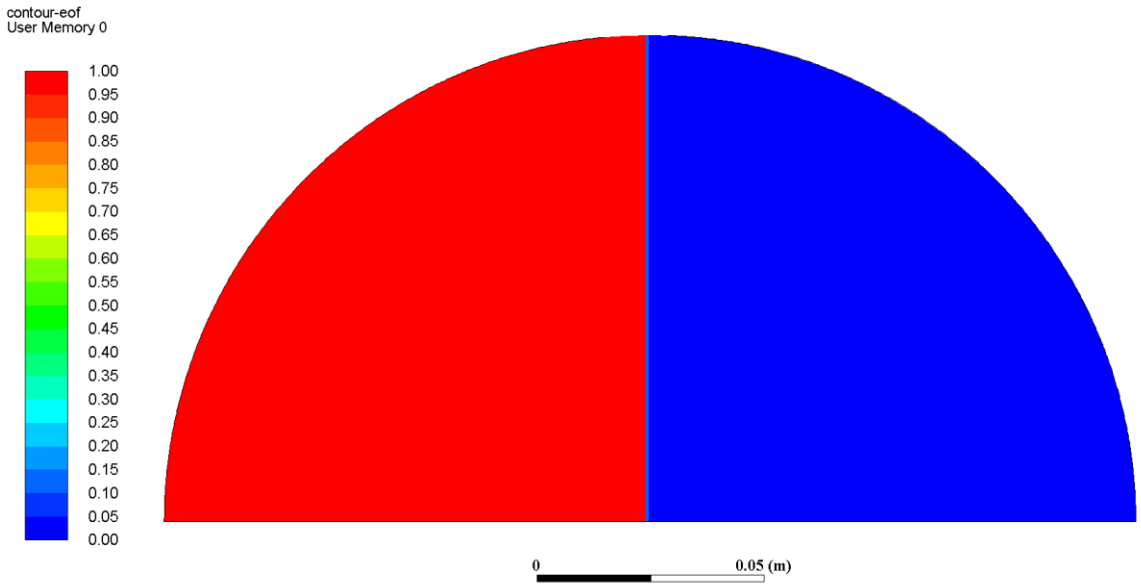


Figure 33 Normal gravity setup: Initial phase fraction distribution.

ii. Results and Discussion

The simulations are set to run to a total flow time of 1,000 sec and at a time step size of 0.01 sec. Because the property data is only valid to 1 MPa, the simulation automatically ends once the pressure reaches this point. As a result, the simulations end at around 680 sec. As expected, the model is mesh sensitive. A mesh convergence study is conducted for the cell sizes shown on Table 12. Each mesh consists of mostly quadrilateral cells with some triangular cells.

Table 12 Normal gravity results: Mesh sizes.

Mesh Size	Cell Count
1.2mm	14,985
1.0mm	21,433
0.8mm	33,579
0.6mm	59,447

Figure 34 shows the transient pressure data of the different cell sizes. The predictions are very close to each other, like in the low gravity case. The differences only appear when the interface location moves across a column of cells. The time at which this occurs depends on two mesh characteristics: (1) the initial phase fraction value at the interface and (2) the average cell size. Since the interface moves to the left, meshes with lower initial phase fraction values will experience the jump earlier. The rate at which the interface moves is assumed to be similar across all mesh sizes. The finer sized meshes will experience the jump earlier for having a shorter travel distance for the interface.

The same sum of squares equation used for the low gravity case is used with this case to judge mesh convergence. The result of this analysis is shown in Table 13. As the data sets are already relatively close together, any variances are increased. This reason may explain the apparent increase in relative error from the 0.8mm mesh to the 0.6mm mesh. Despite this inconsistency, the error mostly decreases as the mesh is refined, suggesting a trend towards mesh

independency. The extent of the mesh refinement was not investigated due to time constraints. Timestep convergence was also not investigated for the same reason.

Table 13 Normal gravity results: Mesh convergence study.

Mesh Size Comparison	Sum of Squared Residuals
1.2mm – 1.0mm	5.05E+12
1.0mm – 0.8mm	4.62E+11
0.8mm – 0.6mm	5.06E+12

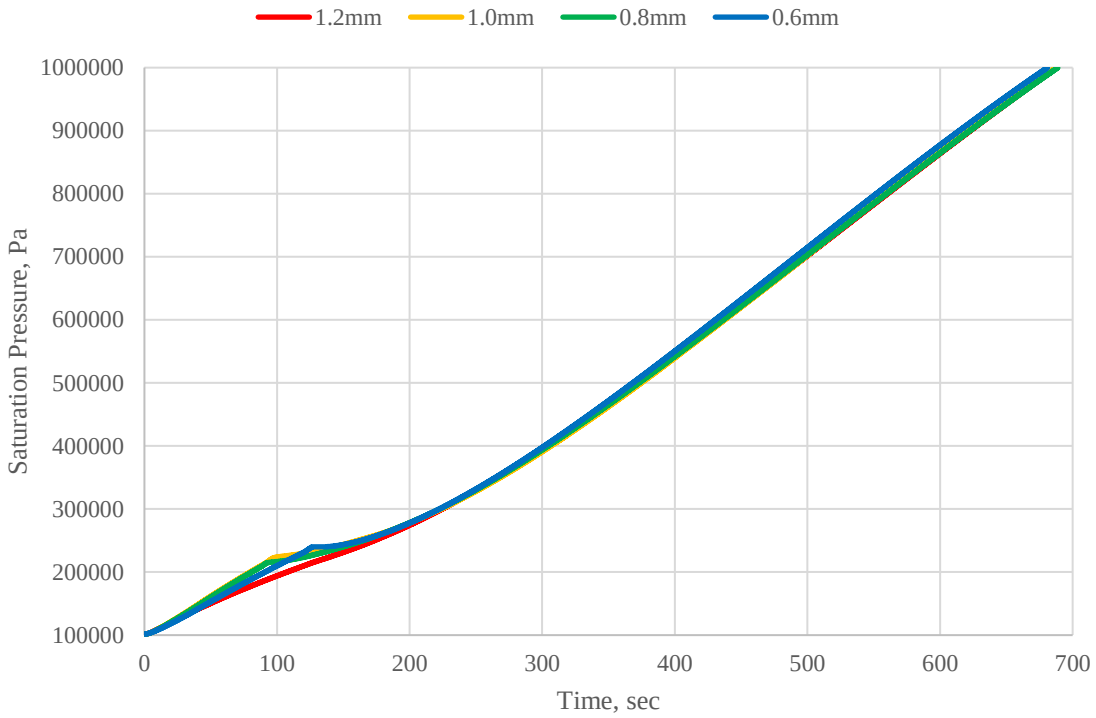
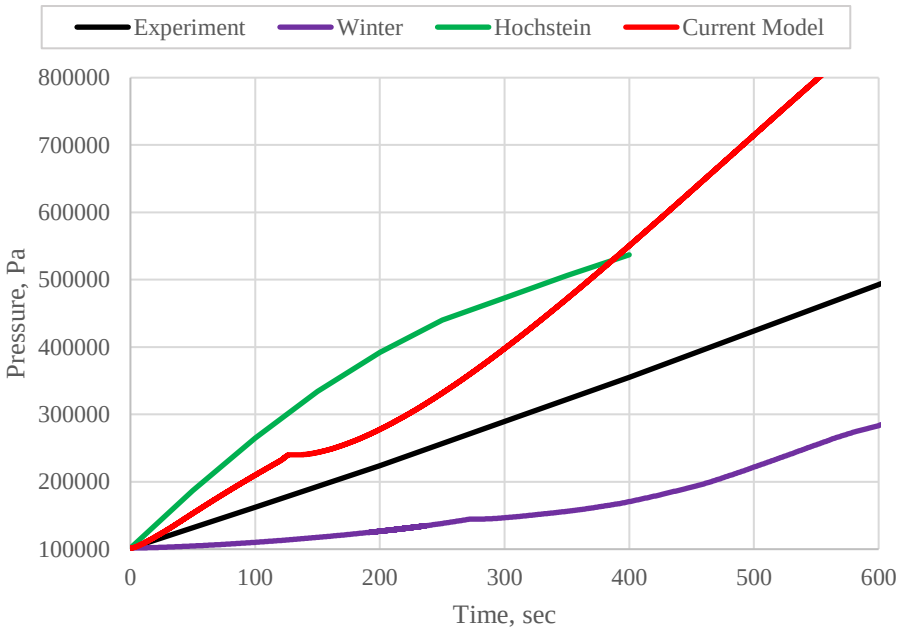


Figure 34 Normal gravity results: Mesh convergence study.

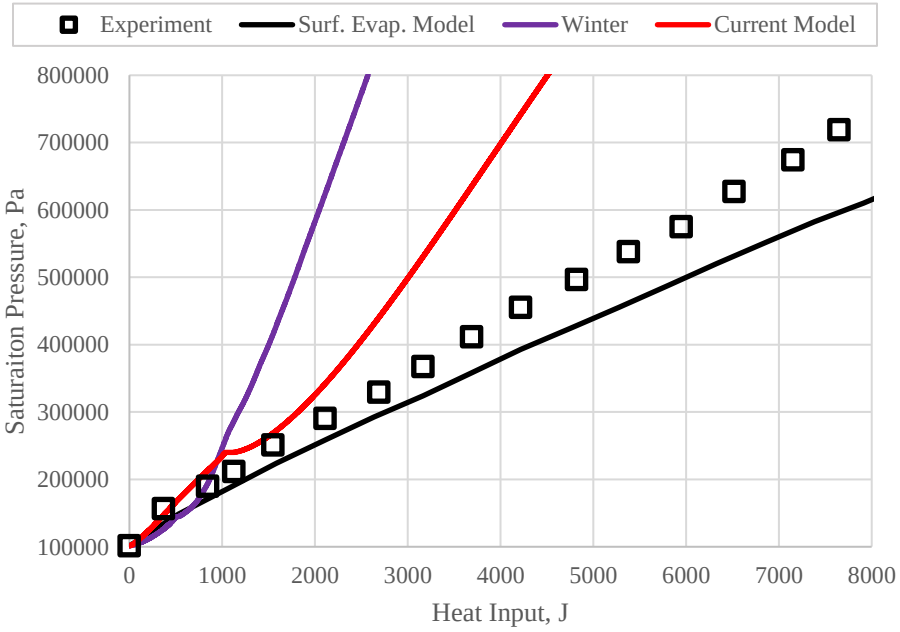
All data shown from this point on are obtained using the finest mesh available. Figure 35 shows comparisons of the new model with the original Winter model, the Hochstein model, and the results obtained by Aydelott. Each prediction has a different shape from the others. These discrepancies may be due to the extreme nature of the experiment. At a pressurization rate of 688 Pa/s, the normal gravity case pressurizes approximately 50 times faster than the low gravity case.

The absence of the phase flux term may be the reason for the discrepancy between the prediction and experiment results. The heat transfer near the interface may not be adequately modeled without the phase flux term.

The extremely high pressure rise rate also affects the interface reconstruction and velocity suppression methods. When the phase fractions of a column of mixed phase cells approaches a pure phase, the adjacent column of cells forms a new wall for the interface to reside. Both walls are maintained until the interface moves to the new column of cells, and the previous column is fully condensed/evaporated. At higher pressurization rates, the time required to maintain both walls is increased before the interface moves across columns. During this time, velocities which were originally adjacent to the interface are zero, leading to a decrease in convective heat transfer to the interface. The resulting pressure rise experiences a sharp change in slope.



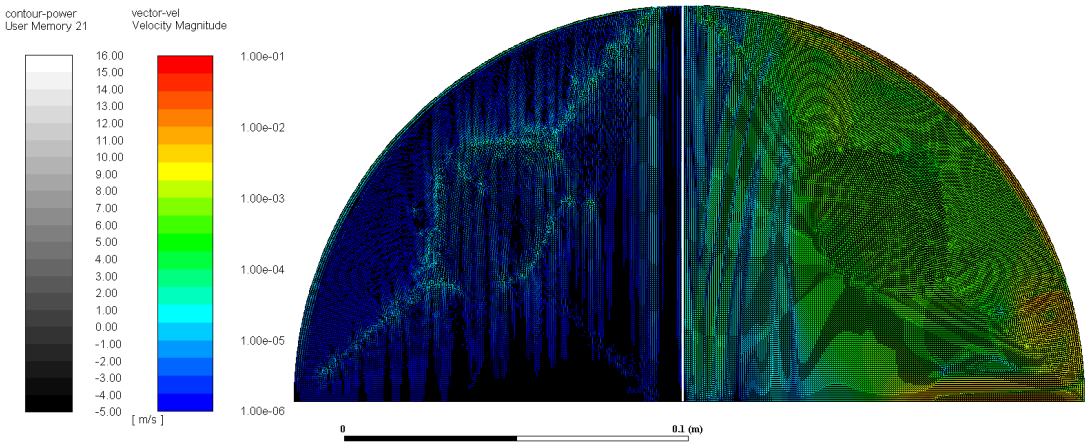
(a)



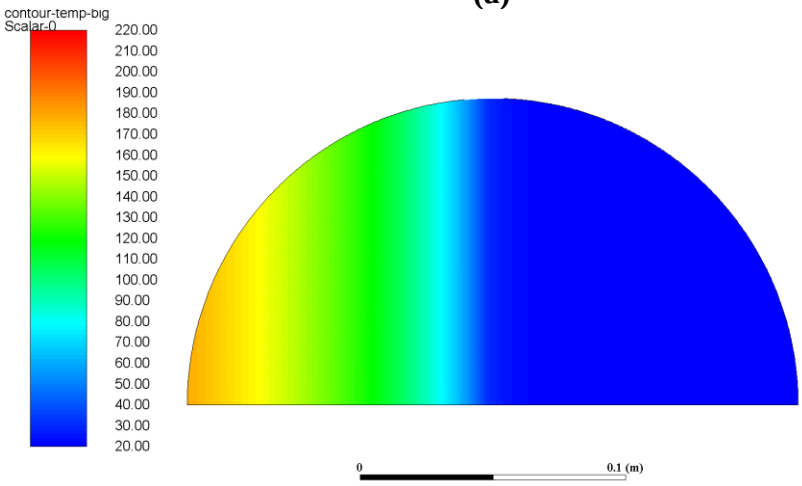
(b)

Figure 35 Normal gravity results: Comparison with previous works with respect to (a) time and (b) heat input.

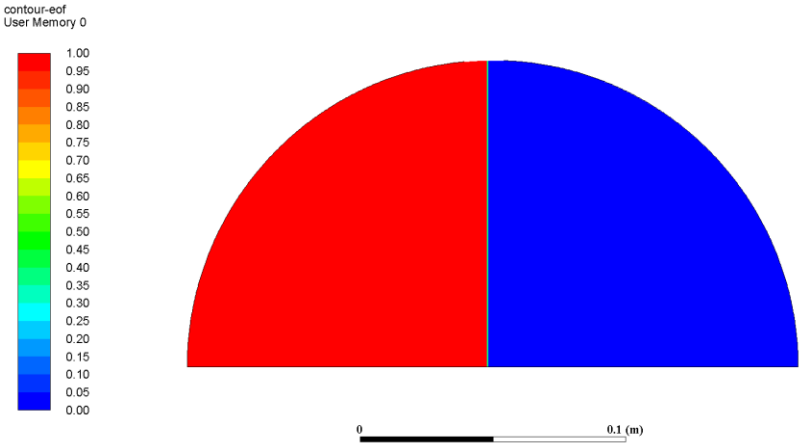
Figure 36 shows the last recorded distributions of the velocity vectors, temperatures, and phase fractions. As previously mentioned in the Computational Methods section, the property data is valid for pressures up to 1 MPa. Afterward, the simulation automatically stops. The presented figures are animation frames saved at 681 sec, before the pressure reached 1 MPa. The ullage region is completely stratified, indicated by the low velocity magnitudes and the axially graduated temperature distribution. The liquid region shows stratification near the interface and convection currents near the walls and the bottom of the tank. A conduction-based model would produce no velocities and a radially graduated temperature distribution. Distributions depicted by the current model are only possible by including natural convection.



(a)



(b)



(c)

Figure 36 Normal gravity results: Final distributions for (a) velocity magnitude, (b) temperature, and (c) phase fraction.

Figure 37a shows that the interface moves across two cell boundaries, as opposed to once in the low gravity case. The first jump occurs early in the simulation, at around 5 sec. The second jump occurs at 106 sec. Figure 37b shows the ELF during the simulation. Like in the low gravity case, the tank begins with a net condensation rate until it stratifies at around 50 sec. Afterwards, heat begins to evaporate liquid at the interface. Figure 37c shows that the temperature rise becomes quasi-steady state. The stratified ullage region is cooled by the liquid and heated by the outer walls, resulting in a semi-equilibrium. Figure 38 depicts the interface reconstruction.

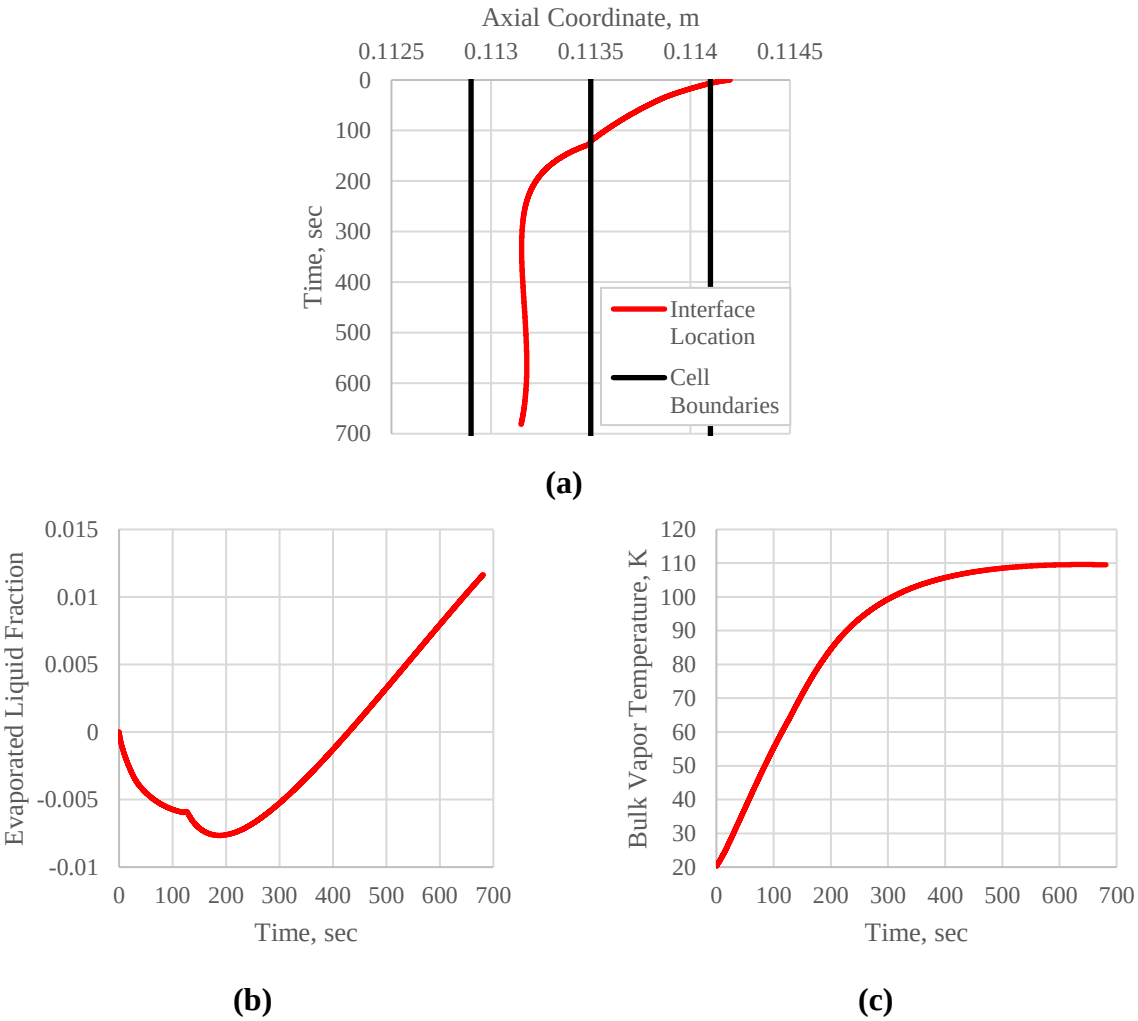


Figure 37 Normal gravity results: (a) interface location, (b) evaporated liquid fraction, (c) bulk vapor temperature.

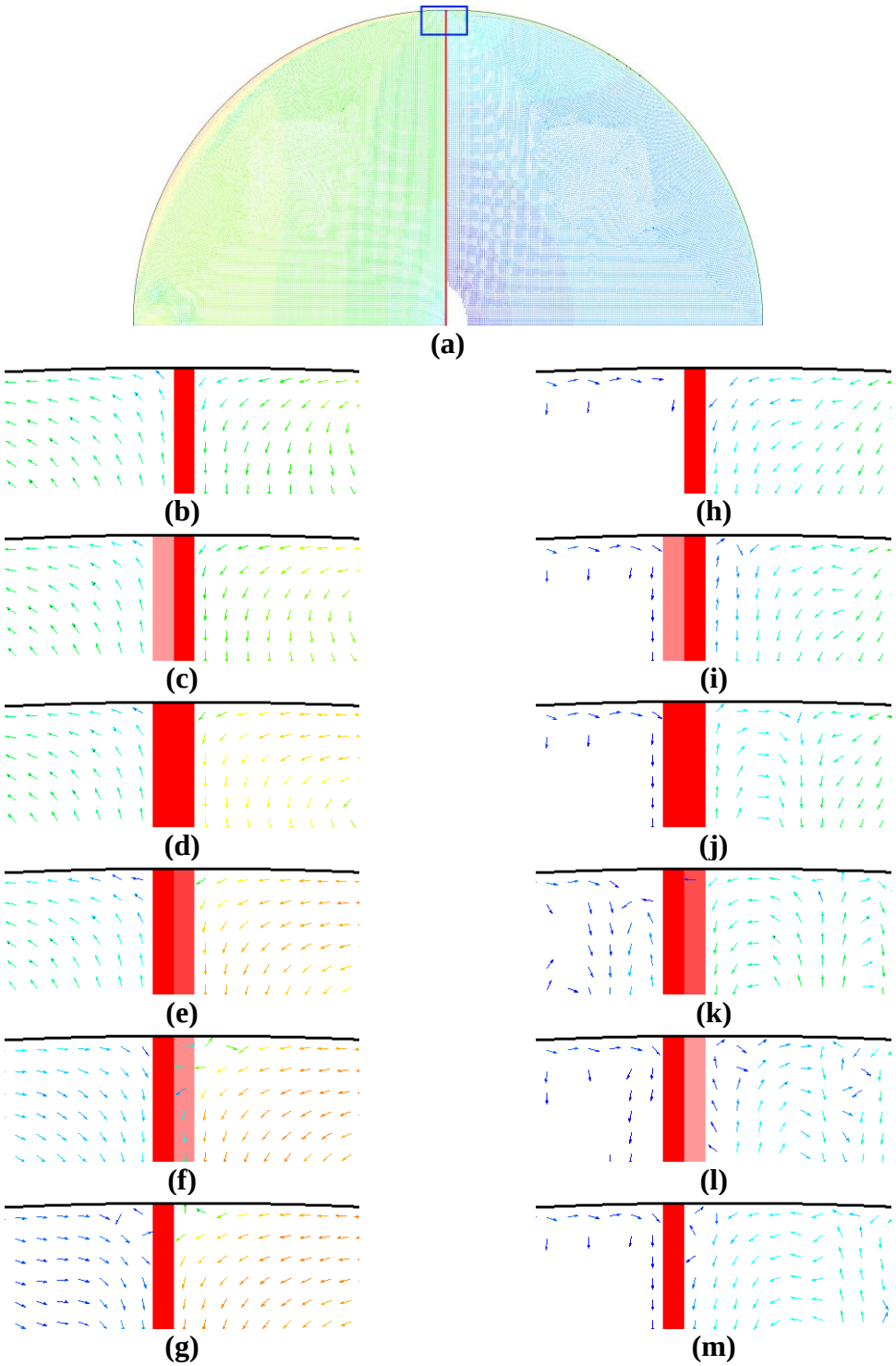


Figure 38 Normal gravity results: Demonstration of interface reconstruction

(a) Vectors of velocity and location of animation frames marked by blue box

at time = (b) 2 sec, (c) 3 sec, (d) 5 sec, (e) 8 sec, (f) 18 sec, (g) 28 sec,

(h) 104 sec, (i) 105 sec, (j) 106 sec, (k) 128 sec, (l) 138 sec, (m) 148 sec.

Figure 39 shows a comparison between the simulation and experiment in terms of the transient temperature distribution of the vapor region. The solid lines denote the simulated temperatures. The dashed lines denote the experiment temperatures. The initial conditions of the simulation do not exactly match with the experiment. Since the experiment begins superheated, the resulting pressure rise is lower. The simulation begins at saturation, which means the increase in temperature consequential to reaching thermal stratification causes a higher rate of pressurization.

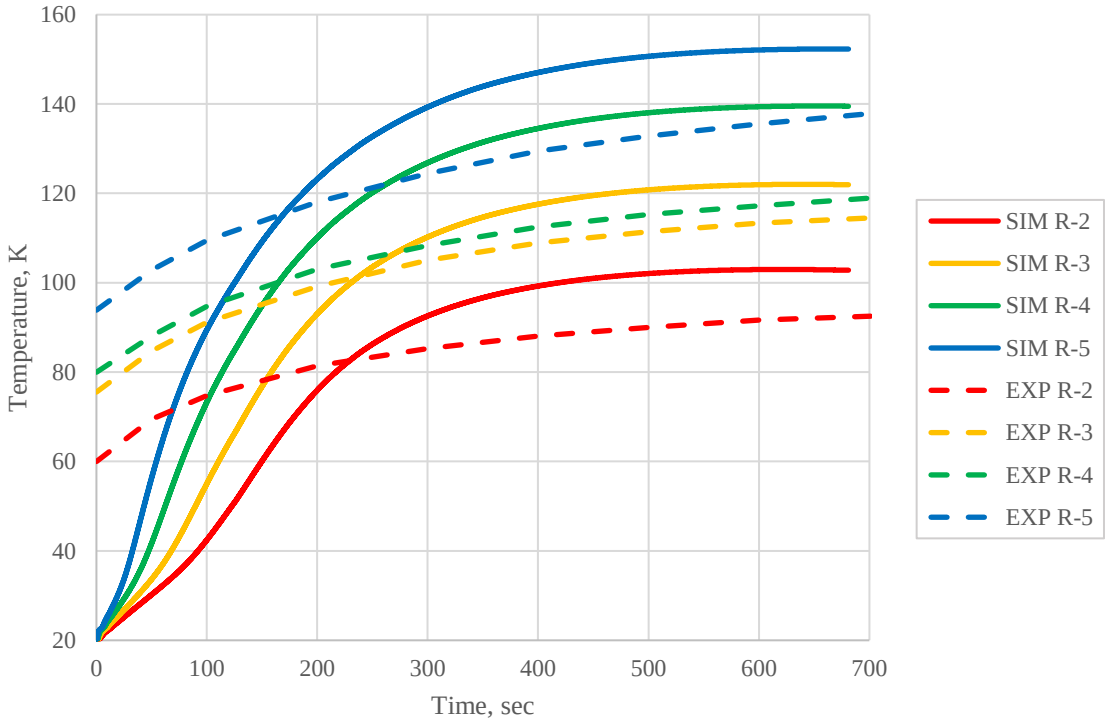


Figure 39 Normal gravity results: Comparison of vapor region temperatures.

SUMMARY AND CONCLUSIONS

In this work, a numerical model is developed to simulate the bulk evaporation and condensation of cryogenic propellants. Finite volume based CFD software Fluent is enhanced by adding an EOF model, which uses an internal energy formulation for conjugate heat transfer and phase change. An implementation of an EOF model developed by Winter is significantly improved. The current model features convection heat transfer, a more comprehensive thermodynamic property model, and a two-phase immiscibility method. With the addition of fluid flow to the heat transfer model, thermal stratification can be accurately simulated within the cryogenic tanks. The numerical model is validated using two experimental cases observing the self-pressurization of LH2 tanks.

The first validation case is the closed fuel tank self-pressurization experiment conducted in low gravity conditions onboard the vehicle AS-203 during orbit. The pressurization rate is largely dependent on the temperature, indicated by the consistently rising bulk vapor temperature and the low ELF variation. The model adequately predicts the pressure rise of the experiment. The final pressure is calculated to have an error of 1.7% to the experiment value. The largest difference in pressure between the experiment and simulation is calculated to be 6.8%. The differences in ullage temperatures between the experiment and simulation range from 1.31% to 32.7%.

The second validation case is a terrestrial study of self-pressurization of a small spherical LH2 tank under normal gravity. The discrepancies with the model predictions may be due to the extreme nature of the pressure rise, which is 50 times faster than the low gravity case. A normal gravity case may need to be investigated with a more gradual rate of pressurization. Another reason may be the inadequacies with convection modeling near the interface. The phase flux

term, which was not implemented into the governing equation for complexity reasons, may be required to adequately model the heat transfer at the interface. Under normal gravity conditions, the resulting fluid velocities are much greater than in low gravity conditions. As a result, the missing convection flux term may be more significant to the governing equation than previously assumed.

The fluid mass is tracked for both validation cases. Since the EOF method is like the VOF method such that the interface is smeared rather than sharp, conservation of mass is not guaranteed. The vapor mass is calculated using the cell densities of the vapor phase, instead of Eq. (25). The low gravity case shows an increase in system mass by about 62 kg, or 0.8%. The normal gravity case shows an increase in system mass by 1.7 g, or 0.7%. Similar to VOF, this change in mass, which is accumulated during the simulation, does not significantly affect the simulation predictions.

REFERENCES

[1] W. D. Ward, L. E. Toole, C. A. Ponder, M. E. Meadows, C. W. Simmons, J. H. Lytle, J. M. McDonald and B. M. Kavanaugh, "Evaluation of AS-203 Low Gravity Orbital Experiment," NASA CR 94045, 1967.

[2] R. D. Bradshaw, "Evaluation and Application of Data from Low-Gravity Orbital Experiment," NASA CR 109847, 1970.

[3] A. Winter, "Simulating Self-Pressurization in Propellant Tanks using an Energy of Fluid Approach," PhD Dissertation, The University of Memphis, 2014.

[4] J. C. Aydelott, "Normal Gravity Self-Pressurization of 9-inch (23 cm) Diameter Spherical Liquid Hydrogen Tankage," NASA TN D-4171, 1967.

[5] J. I. Hochstein, H. Ji and J. C. Aydelott, "Prediction of Self-Pressurization Rate of Cryogenic Propellant Tankage," *Journal of Propulsion and Power,* vol. 6, no. 1, pp. 11-17, 1990.

[6] S. Anghaie and Z. Ding, "Modeling of Bulk Evaporation and Condensation," NASA CR 198392, 1996.

[7] S. Anghaie and Z. Ding, "Thermal-hydraulic Analysis of Bulk Evaporation and Condensation in a Multiphase Nuclear Fuel Cell," *Nuclear Technology*, vol. 120, pp. 57-70, 1997.

[8] Z. Ding and S. Anghaie, "Numerical Modeling of Conduction-driven Bulk Evaporation and Condensation Processes with Constant Volume," *International Journal for Numerical Methods in Engineering,* vol. 39, pp. 219-233, 1996.

[9] "Thermophysical Properties of Parahydrogen," National Institute of Standards and Technology, 2021. [Online]. Available: https://webbook.nist.gov/cgi/cbook.cgi?Name=parahydrogen&Units=SI.

[10] D. E. Daney, "Turbulent Natural Convection of Liquid Deuterium, Hydrogen, and Nitrogen within Enclosed Vessels," NBSIR 75-807, 1975.

[11] ANSYS Fluent 19.1, 2018. [Online].

[12] ANSYS Fluent 19.1, "User's Guide," 2021. [Online].

[13] ANSYS Fluent 19.1, "Customization Manual," 2021. [Online].

[14] Y. A. Cengel and M. A. Boles, "Thermodynamic Property Relations," in *Thermodynamics: An Engineering Approach*, 8th ed., New York, McGraw-Hill Education, 2015, pp. 661-668.

[15] W. C. Reynolds, *Thermodynamic Properties in SI,* Stanford, CA: Stanford Univeristy Department of Mechanical Engineering, 1979, pp. 122-125,139.

[16] B. A. Younglove, "Thermophysical properties of fluids. I. Argon, ethylene, parahydrogen, nitrogen, nitrogen trifluoride, and oxygen," *Journal of Physical and Chemical Reference Data,* vol. 11, 1982.

[17] J. T. Clark, "Calculation of Thermodynamic Properties of Hydrogen using FORTRAN 90/95," Master's Project Report, The University of Memphis, 2002.

[18] N. E. Todreas and M. S. Kazimi, "Transport Equations for Two-Phase Flow," in *Nuclear Systems I: Thermal Hydraulic Fundamentals*, Hemisphere Publishing Corporation, 1990, p. 133.

[19] ANSYS Fluent 19.1, "Theory Guide," 2021. [Online].

[20] S. V. Patankar, "Source Term Linearization," in *Numerical Heat Transfer and Fluid Flow*, Hemisphere Publishing Corporation, 1980, pp. 143-148.

[21] W. J. Masica, "Exploring in Aerospace Rocketry: 8. Zero-Gravity Effects," NASA TM X-52395, 1966-1967.

[22] J. A. Salzman and W. J. Masica, "Lateral Sloshing in Cylinders under Low-Gravity Conditions," NASA TN D-5058, 1969.

[23] T. A. Coney and J. A. Salzman, "Lateral Sloshing in Oblate Spheroidal Tanks under Reduced- and Normal-Gravity Conditions," NASA TN D-6250, 1971.

[24] G. E. Myers, *Analytical Methods in Conduction Heat Transfer*, 2 ed., Madison, WI: AMCHT Publications, 1998, p. 135.

[25] H. S. Carslaw and J. C. Jaeger, *Conduction of Heat in Solids*, 2 ed., New York, New York: Oxford Press, 1959, pp. 286-287.

APPENDIX A FLUENT SETUP FOR UNSTEADY CONDUCTION CASES

This appendix details the setup procedure for the two unsteady conduction verification cases in the Unsteady Conduction Heat Transfer section. The source code for these cases is provided in Appendix B.

1. Start *Fluent*.
    a. Under *Dimension*, select *2D*.
    b. Under *Processing Options*, choose either *Serial* or *Parallel*.
    c. If *Parallel* is chosen, use a number of processes that is under the total number of logical processors in your system.
    d. If using *Fluent* in a *High-Performance Computing* unit, the number of logical processors available will depend on the number of processors assigned to your job.
    e. Under *Working Directory*, ensure this is the folder where you will read and write files for your simulation. The source codes and mesh file will need to be in this folder as well.
    f. Press *OK* to start *Fluent*.
2. Read the 0.1m x 0.1m 100x100 cell mesh. *File* → *Read* → *Mesh…*
    a. If the 100x100 cell mesh was not generated for a 0.1m x 0.1m geometry, In the *Tree, Setup* → *General*. Press the *Scale…* option to scale the mesh to the appropriate dimensions.
3. In the *Tree, Setup* → *General*. Select the *Transient* option under *Solver* → *Time*.
4. Compile the UDF library.
    a. In the ribbon, *User Defined* → *Functions* → *Compiled…* to open the *Compiled UDFs* dialog box.
    b. Under *Source Files*, select *Add…* and select the source code and press *OK*.
    c. Click on *Build* in the *Compiled UDFs* dialog box. If there are no errors, click on *Load*.
5. Create the UDS.
    a. In the ribbon, *User Defined* → *Scalars…* to open the *User-Defined Scalars* dialog box.
    b. Set the *Number of User-Defined Scalars* to 1.
    c. Set the *Flux Function* to the *none* and the *Unsteady Function* to *uds_unsteady*.
6. Create the fluid for water.
    a. In the *Tree, Setup* → *Materials* → *Fluid* → *air* → *Edit…*
    b. For *Density*, replace the constant value with the one found in Table 6.
    c. For *UDS Diffusivity*, press *Edit…* and replace the constant value with the one found in Table 6. Press *OK*.
    d. Press *Change/Create* to save the changes and close the *Create/Edit Materials* dialog box.
7. Assign the boundary conditions.
    a. In the *Tree, Setup* → *Boundary Conditions*. Open the respective *Wall* dialog box and go to the *UDS* tab.
    b. For a temperature boundary condition, use *Specified Value* from the dropdown menu under *User-Defined Scalar Boundary Condition*.

    c. For a heat flux boundary condition, use *Specified Flux*.
    d. Set the boundary value according to the specifications given in the Unsteady Conduction Heat Transfer section.
    e. For adiabatic walls, use a *Specified Flux* with a boundary value of 0.
8. Initialize the solution.
    a. In the *Tree, Solution → Initialization*.
    b. Set the *Initial Value* of *User Scalar 0* to 300 K. Press *Initialize*.
9. Set the transient calculation options.
    a. In the *Tree, Solution → Run Calculation* to activate the *Run Calculation Task Page*.
    b. Choose a *Time Step Size* of 1 sec and set the *Number of Time Steps* to 600.
    c. Set *Max Iterations/Time Step* to at least 100.
    d. Leave the *Reporting Interval* as 1.
10. Run the simulation.
    a. In the *Tree, Solution → Run Calculation*.
    b. Press *Calculate* to run.

APPENDIX B SOURCE CODE FOR UNSTEADY CONDUCTION CASES

```
#include "udf.h"

DEFINE_UDS_UNSTEADY(uds_unsteady, c, t, i, apu, su)
{
        real physical_dt, vol, rho, phi_old;
        physical_dt = RP_Get_Real("physical-time-step");
        vol = C_VOLUME(c, t);
        rho = C_R_M1(c, t);
        real Cv = 4130.0;
        *apu = -rho * vol * Cv / physical_dt;/*implicit part*/
        phi_old = C_STORAGE_R(c, t, SV_UDSI_M1(i));
        *su = rho * vol * Cv * phi_old / physical_dt;/*explicit part*/
}
```

APPENDIX C FLUENT SETUP FOR UNSTEADY CONDUCTION PHASE CHANGE CASE

This appendix details the setup of the case found in the Unsteady Conduction Phase Change section. The source code is provided in Appendix D.

1. Start *Fluent*.
    a. Under *Dimension*, select *2D*.
    b. Under *Processing Options*, choose either *Serial* or *Parallel*.
    c. If *Parallel* is chosen, use a number of processes that is under the total number of logical processors in your system.
    d. If using *Fluent* in a *High Performance Computing* unit, the number of logical processors available will depend on the number of processors assigned to your job.
    e. Under *Working Directory*, ensure this is the folder where you will read and write files for your simulation. The source codes and mesh file will need to be in this folder as well.
    f. Press *OK* to start *Fluent*.
2. Read the 0.0025m x 0.1m mesh. *File* → *Read* → *Mesh…*
3. If the mesh was not generated for a 0.0025m x 0.1m geometry, In the *Tree*, *Setup* → *General*. Press the *Scale…* option to scale the mesh to the appropriate dimensions.
4. In the *Tree*, *Setup* → *General*. Select the *Transient* option under *Solver* → *Time*.
5. Compile the UDF library.
    a. In the ribbon, *User Defined* → *Functions* → *Compiled…* to open the *Compiled UDFs* dialog box.
    b. Under *Source Files*, select *Add…* and select the source code and press *OK*.
    c. Click on *Build* in the *Compiled UDFs* dialog box. If there are no errors, click on *Load*.
6. Create the UDS.
    a. In the ribbon, *User Defined* → *Scalars…* to open the *User-Defined Scalars* dialog box.
    b. Set the *Number of User-Defined Scalars* to 1.
    c. Set the *Flux Function* to the *none* and the *Unsteady Function* to *none*.
7. Create the UDMs.
    a. In the ribbon, *User Defined* → *Memory…* to open the *User-Defined Memory* dialog box.
    b. Set the number of *User-Defined Memory Locations* to 10.
8. Attach the function hooks.
    a. In the ribbon, *User Defined* → *Function Hooks…* to open the *User-Defined Function Hooks* dialog box.
    b. For *Initialization*, *Adjust*, and *Execute at End* press *Edit…* and add the respective function hooks.
9. Create the fluid for water.
    a. In the *Tree*, *Setup* → *Materials* → *Fluid* → *air* → *Edit…*
    b. For *Density* and *UDS Diffusivity*, choose *user-defined* in the dropdown menu and select the respective property functions.

c. Press *Change/Create* to save the changes and close the *Create/Edit Materials* dialog box.

10. Setup the source terms for the simulation.
    a. In the *Tree, Setup → Cell Zone Conditions → fluid → Edit…*
    b. Select the *Source Terms* option and go to the *Source Terms* tab.
    c. Press *Edit…* to add source terms.
    d. For *User Scalar 0*, set *Number of User Scalar 0 sources* to 2 and add the unsteady term and the latent heat term.
    e. Press *OK* to leave the dialog boxes and go back to the main window.
11. Create the *timestep-ender* report definition.
    a. In the *Tree, Solution → Report Definitions → New → User Defined…* to open the *User Defined Report Definition* dialog box.
    b. Name the report definition *timestep-ender* and choose *report_iter* from the *Function* dropdown menu.
12. Setup the timestep convergence criteria.
    a. In the *Tree, Solution → Monitors → Residual → Edit…* to open the *Residual Monitors* dialog box.
    b. Uncheck the *Print to Console* and *Plot* options under *Options*.
    c. Uncheck the *Check Convergence* options for all of the equations.
    d. Press *Convergence Conditions…* to open the *Convergence Conditions* dialog box.
    e. Select the *Time Step Convergence* option to add a convergence condition.
    f. Choose the *timestep-ender* in the *Report Definition* dropdown menu.
    g. Set the *Stop Criterion* to a very low value (1e-24) to prohibit unwanted activation.
    h. Press *OK* until back at the main window.
13. Assign the boundary conditions.
    a. In the *Tree, Setup → Boundary Conditions*. Open the respective *Wall* dialog box and go to the *UDS* tab.
    b. Use *Specified Value* from the dropdown menu under *User-Defined Scalar Boundary Condition*. Set the boundary values of the top and bottom walls according to the specifications given in the

    d. Unsteady Conduction Phase Change section.
    e. For adiabatic walls, use a *Specified Flux* with a boundary value of 0.
14. Create the rest of the report definitions.
    a. In the *Tree, Solution* → *Report Definitions* → *New* → *User Defined…* to open the *User Defined Report Definition* dialog box.
    b. Name the report definitions according to their names from the UDF.
    c. To compare with Eqs. (36) and (37), *report_yp* $= Y$.
15. Create report files to be output during the simulation.
    a. In the *Tree, Solution* → *Monitors* → *Report Files* → *New…* to create a new report file.
    b. Add all desired report definitions and press *Browse…* under *Output File Base Name*.
    c. Set the desired filename and include the .txt extension.
    d. Press *OK* until back to the main window.
16. Initialize the solution.
    a. In the *Tree, Solution* → *Initialization*.
    b. Set the *Initial Value* of *User Scalar 0* to 275.15 K.
    c. Press *Initialize*.
17. Set the transient calculation options.
    a. In the *Tree, Solution* → *Run Calculation* to activate the *Run Calculation Task Page*.
    b. Choose a *Time Step Size* of 1 sec.
    c. Set the *Number of Time Steps* to 600.
    d. Set *Max Iterations/Time Step* to at least 100.
    e. Leave the *Reporting Interval* as 1.
18. Run the simulation.
    a. In the *Tree, Solution* → *Run Calculation*.
    b. Press *Calculate* to run.

APPENDIX D SOURCE CODE FOR UNSTEADY CONDUCTION PHASE CHANGE CASE

```
#include "udf.h"

/* Thread Indices */
int fluid = 4;
int wall_side = 5;
int wall_top = 6;
int wall_bottom = 7;

/* User-defined Scalars */
#define T(c, t) C_UDSI(c, t, 0)
#define FT(f, t) F_UDSI(f, t, 0)

/* User-defined Memory Variables */
#define e(c, t) C_UDMI(c, t, 0)
#define eof(c, t) C_UDMI(c, t, 1)
#define rho(c, t) C_UDMI(c, t, 2)
#define cv(c, t) C_UDMI(c, t, 3)
#define k(c, t) C_UDMI(c, t, 4)

/* Stored UDMs */
#define e_sen(c, t) C_UDMI(c, t, 5)
#define e_sen_m1(c, t) C_UDMI(c, t, 6)
#define eof_m1(c, t) C_UDMI(c, t, 7)
#define eof_old(c, t) C_UDMI(c, t, 8)
#define e_m1(c, t) C_UDMI(c, t, 9)

double Tsat = 275.15, deltae = 334944.;
double eL, eS;
double rhoL = 1000., rhoS = 920.;
double cvL = 4186.8, cvS = 2101.8;
double kL = 0.6029, kS = 2.2190;

double resid, residual_uds_0, residual_eof;
double tolerance_uds_0 = 1e-9, tolerance_eof = 1e-6;
int iter, res_count, flag;
double mix, ymix, yp;

/*
The sensible internal energy is a function of temperature
e = integral[cv(T)dT]
de/dT = cv(T)
*/
DEFINE_INIT(hook_initialization, d)
{
#if !RP_HOST
        Thread *t = Lookup_Thread(d, fluid);
        Thread *t_side = Lookup_Thread(d, wall_side);
        Thread *t_top = Lookup_Thread(d, wall_top);
        Thread *t_bottom = Lookup_Thread(d, wall_bottom);
        cell_t c;

        Message0("\nBeginning Initialization...\n");

        iter = 0;
        residual_uds_0 = 0;
        residual_eof = 0;

        begin_c_loop(c, t)
        {
                eof(c, t) = 1;
                eof_m1(c, t) = eof(c, t);
        }
        end_c_loop(c, t)

        begin_c_loop(c, t)
        {
                if (eof(c, t) == 0) e(c, t) = cvS * T(c, t);
                if (0 < eof(c, t) && eof(c, t) < 1) e(c, t) = cvS * Tsat + eof(c, t)*deltae;
                if (eof(c, t) == 1) e(c, t) = cvL * (T(c, t) - Tsat) + deltae + cvS * Tsat;

                e_sen(c, t) = e(c, t) - eof(c, t)*deltae;

                /* Material Properties */
                rho(c, t) = eof(c, t)*rhoL + (1 - eof(c, t))*rhoS;
                cv(c, t) = eof(c, t)*cvL + (1 - eof(c, t))*cvS;
                k(c, t) = eof(c, t)*kL + (1 - eof(c, t))*kS;

                e_sen_m1(c, t) = e_sen(c, t);
                e_m1(c, t) = e(c, t);
        }
        end_c_loop(c, t)

        Message0("Initialization Complete! :^)\n");
#endif
        node_to_host_int_1(iter);
}

DEFINE_ADJUST(hook_adjust, d)
{
#if !RP_HOST
        Thread *t = Lookup_Thread(d, fluid);
        cell_t c;

        /* Reset residuals */
        if (iter == 0)
        {
                residual_eof = 0;
                res_count = 0;
        }

        /* Get fluent residuals */
        if (iter > 1)
        {
                /* temperature */
                if (DOMAIN_RES_SCALE(d, 3) == 0) residual_uds_0 = DOMAIN_RES(d, 3);
                else residual_uds_0 = DOMAIN_RES(d, 3) / DOMAIN_RES_SCALE(d, 3);
        }
```

```
PRF_GSYNC();
iter++;
flag = 0;

/* Update eof */
begin_c_loop(c, t)
{
	eS = cvS * Tsat;
	eL = eS + deltae;

	if (eof(c, t) == 0) e(c, t) = cvS * T(c, t);
	if (0 < eof(c, t) && eof(c, t) < 1) e(c, t) = cvS * T(c, t) + eof(c, t)*deltae;
	if (eof(c, t) == 1) e(c, t) = cvL * (T(c, t) - Tsat) + deltae + cvS * Tsat;

	e_sen(c, t) = e(c, t) - eof(c, t)*deltae;

	/* Save old eof for residual */
	eof_old(c, t) = eof(c, t);

	/* Calculate new eof */
	if (e(c, t) <= eS) eof(c, t) = 0;
	if (e(c, t) >= eL) eof(c, t) = 1;
	if (e(c, t) > eS && e(c, t) < eL) eof(c, t) = (e(c, t) - eS) / (eL - eS);

}
end_c_loop(c, t)

/* Calculate eof residual */
residual_eof = 0;
begin_c_loop(c, t)
{
	/* Cell of maximum change */
	if (eof_old(c, t) == 0) resid = eof(c, t);
	else resid = fabs((eof(c, t) - eof_old(c, t)) / eof_old(c, t));
	if (resid > residual_eof) residual_eof = resid;
}
end_c_loop(c, t)
residual_eof = PRF_GRHIGH1(residual_eof);
if (residual_eof < tolerance_eof) flag = 1;

/* Update Properties */
begin_c_loop(c, t)
{
	rho(c, t) = eof(c, t)*rhoL + (1 - eof(c, t))*rhoS;
	cv(c, t) = eof(c, t)*cvL + (1 - eof(c, t))*cvS;
	k(c, t) = eof(c, t)*kL + (1 - eof(c, t))*kS;
}
end_c_loop(c, t)

if (residual_uds_0 > tolerance_uds_0) flag = 0;
if (residual_eof > tolerance_eof) flag = 0;

/* Display residual */
if (iter%RP_Get_Integer("iteration-chunk") == 0)
{
	res_count++;
	if (res_count % 20 == 1)
	{
		Message0("\nTime = %g sec\n", CURRENT_TIME);
		Message0("|<----Solution Residuals---->|\n");
		Message0("  iter      energy         eof\n");
	}
	Message0("%6d  %10.4e  %10.4e\n", iter, residual_uds_0, residual_eof);
}

/* End time step */
if (flag == 1 && iter != 1 && iter%RP_Get_Integer("iteration-chunk") == 0)
{
	Message0("Solution converged in %d iterations\n", iter);
	Message0("    energy         eof\n");
	Message0("%10.4e  %10.4e\n", residual_uds_0, residual_eof);
	iter-=RP_Get_Integer("iteration-chunk");
}
#endif
	node_to_host_real_2(residual_uds_0, residual_eof);
	node_to_host_int_1(iter);
}

DEFINE_UDS_UNSTEADY(uds_unsteady, c, t, i, apu, su)
{
	real dt, phi_old;
	dt = CURRENT_TIMESTEP;
	*apu = -rho(c, t) * C_VOLUME(c, t) * cv(c, t) / dt;/*implicit part*/
	phi_old = C_STORAGE_R(c, t, SV_UDSI_M1(i));
	*su = rho(c, t) * C_VOLUME(c, t) * cv(c, t) * phi_old / dt;/*explicit part*/
}

DEFINE_SOURCE(source_unsteady_total_heat, c, t, dS, eqn)
{
	real source;
	real dt = CURRENT_TIMESTEP;
	source = rho(c, t) * (e_m1(c, t) - e(c, t)) / dt;
	dS[eqn] = - rho(c, t) * cv(c, t) / dt;
	return source;
}
DEFINE_SOURCE(source_unsteady, c, t, dS, eqn)
{
	real source;
	real dt = CURRENT_TIMESTEP;
	source = rho(c, t) * (e_sen_m1(c, t) - e_sen(c, t)) / dt;
	dS[eqn] = - rho(c, t) * cv(c, t) / dt;
	return source;
}
DEFINE_SOURCE(source_latent_heat, c, t, dS, eqn)
{
	real dt, source;
	dt = CURRENT_TIMESTEP;
	source = rho(c, t) * deltae * (eof_m1(c, t) - eof(c, t)) / dt;
	dS[eqn] = 0;
	return source;
}

DEFINE_DIFFUSIVITY(property_diffusivity, c, t, i)
{
	real k = k(c, t);
	return k;
```

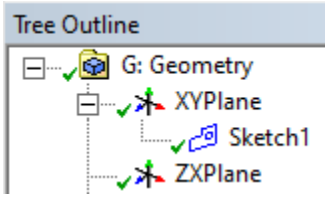
```
}
DEFINE_PROPERTY(property_density, c, t)
{
	real rho = rhoL;
	return rho;
}

DEFINE_EXECUTE_AT_END(hook_execute_at_end)
{
	iter = 0;
#if !RP_HOST
	Domain *d = Get_Domain(1);
	Thread *t = Lookup_Thread(d, fluid);
	cell_t c;
	real x[ND_ND];

	/* Store for next timestep */
	PRF_GSYNC();
	begin_c_loop(c, t)
	{
		e_sen_m1(c, t) = e_sen(c, t);
		e_m1(c, t) = e(c, t);
		eof_m1(c, t) = eof(c, t);
	}
	end_c_loop(c, t)

	/* Record interface location */
	mix = 1;
	ymix = 1;
	real length = 10;
	begin_c_loop_int(c, t)
	{
		C_CENTROID(x, c, t);
		length = sqrt(C_VOLUME(c, t));
		if (-length/4 < x[0] && x[0] < length*3/4)
		{
			if (eof(c, t) < mix && eof(c, t) > 0)
			{
				mix = eof(c, t);
				ymix = x[1];
			}
		}
	}
	end_c_loop_int(c, t)
	mix = PRF_GRLOW1(mix);
	ymix = PRF_GRLOW1(ymix);
	length = PRF_GRLOW1(length);
	yp = ymix + (0.5 - mix) * length;
	if (CURRENT_TIME == 0) yp = 0;
	Message0("%g %g %g\n", ymix, mix, yp);
#endif
	node_to_host_real_3(mix, ymix, yp);
}

DEFINE_REPORT_DEFINITION_FN(report_iter)
{
	node_to_host_int_1(iter);
	return iter;
}
DEFINE_REPORT_DEFINITION_FN(report_mix)
{
	node_to_host_real_1(mix);
	return mix;
}
DEFINE_REPORT_DEFINITION_FN(report_ymix)
{
	node_to_host_real_1(ymix);
	return ymix;
}
DEFINE_REPORT_DEFINITION_FN(report_yp)
{
	node_to_host_real_1(yp);
	return yp;
}
```

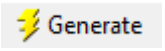
APPENDIX E GEOMETRY AND MESH GENERATION OF VALIDATION CASES

This appendix details the creation of the geometry and mesh of the two validation cases. The programs used in this section are included in the ANSYS Workbench package.

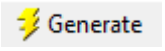
Low Gravity Large Scale Orbit Fuel Tank

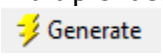
DesignModeler

1. Open ANSYS Workbench.
2. Drag and drop the *Geometry* module from the Toolbox into the *Project Schematic* workspace.
3. Right click on Item 2 *Geometry* and select *New DesignModeler Geometry* to open the DesignModeler program.
4. In DesignModeler, select the *XYPlane* in the *Tree Outline*.
5. Select *New Sketch*  and *Sketch1* will appear under *XYPlane*.


6. Ensure *Sketch1* is selected and move to the *Sketching* tab in the *Tree Outline*.
7. Sketch the exterior tank geometry shown on Figure 40. The dimensions are given in Table 14.
8. All arcs have the same radius. Use *Constraints* in *Sketching Toolboxes* to apply these conditions if they are not applied already.
9. Create a second sketch. Place and dimension the interior lines of the tank shown on Figure 40 using the dimensions on Table 14.
10. Return to the *Modeling* tab and select *Sketch1*.
11. In the top ribbon, *Concept* → *Surface from Sketches*.
12. Ensure *Sketch1* is selected as the *Base Object*.
13. In the ribbon, click on the *Generate* button  to create the surface.

14. In the top ribbon, *Tools* → *Face Split*.
15. Select the new surface as the *Target Face*.
16. Choose the lines from *Sketch2* as the *Tool Geometry*. In the top ribbon, ensure line bodies  are chosen in the selection filter.

17. In the ribbon, click on the *Generate* button  to split the surface.

18. Name the fluid zone and the wall sections. In the top ribbon, *Tools* → *Named Selection*. Multiple faces/edges can be assigned to a single *Named Selection*. Click *Generate*  after each assignment. The vertical edges are in the interior named selection.

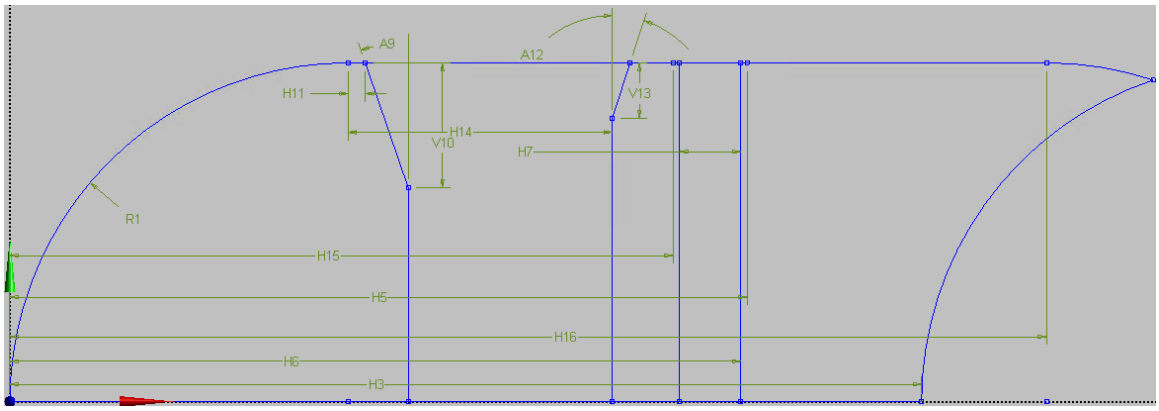


Figure 40 AS203 tank geometry.

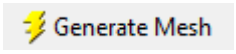
Table 14 AS203 sketch dimensions.

Dimension Name	Value
R1	3.30 m
H3	8.90 m
H5	7.196 m
H6	7.136 m
H7	0.6 m
A9	19°
V10	1.215 m
H11	0.17 m
A12	18°
V13	0.54 m
H14	2.58 m
H15	6.476
H16	10.12

Table 15 AS203 named selections.

Named Selection	Number of Geometries	Faces/ Edges
fluid	5	Faces
axis	5	Edges
interior	4	Edges
wall-fwd	1	Edge
wall-dry	3	Edges
wall-adi	3	Edges
wall-wet	1	Edge
wall-aft	1	Edge
wall-com	1	Edge
wall-def	1	Edge
wall-baf	1	Edge

Meshing

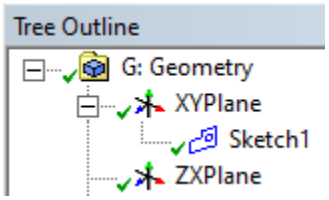
1. Drag and drop the *Mesh* module on top of the *Geometry* module to link the two.
2. Double click on *Mesh* to open the *Meshing* program.
3. In the *Outline*, select *Mesh*.
4. In the window, click the dropdown menu for *Solver Preference* and select *Fluent*.
5. Enter the intended cell size in *Element Size* (Pay attention to the units).
6. Generate the mesh by clicking on *Generate Mesh* 

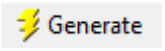
in the top ribbon.
7. Export the mesh. In the top ribbon, *File* → *Export*. Name the file to your preference.

Normal Gravity Small Scale Terrestrial Spherical Tank

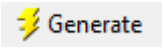
DesignModeler

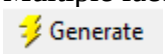
1. Open ANSYS Workbench.
2. Drag and drop the *Geometry* module from the Toolbox into the *Project Schematic* workspace.
3. Right click on Item 2 *Geometry* and select *New DesignModeler Geometry* to open the DesignModeler program.
4. In DesignModeler, select the *XYPlane* in the *Tree Outline*.
5. Select *New Sketch* and *Sketch1* will appear under *XYPlane*.



6. Ensure *Sketch1* is selected and move to the *Sketching* tab in the *Tree Outline*.
7. Sketch the outer semicircle geometry shown on Figure 41 using the tools in the *Draw* menu of the *Sketching Toolboxes* window.
8. All arcs are concentric and have the same diameter of 230mm. Use *Constraints* in *Sketching Toolboxes* to apply these conditions if they are not applied already.
9. Create a second sketch. Place and dimension the vertical lines shown on Figure 41 using the dimension values in Table 16.
10. Return to the *Modeling* tab and select *Sketch1*.
11. In the top ribbon, *Concept* → *Surface from Sketches*.
12. Ensure *Sketch1* is selected as the *Base Object*.
13. In the ribbon, click on the *Generate* button 

to create the surface.
14. In the top ribbon, *Tools* → *Face Split*.
15. Select the new surface as the *Target Face*.
16. Choose the two lines from *Sketch2* as the *Tool Geometry*. In the top ribbon, ensure line bodies 

are chosen in the selection filter.
17. In the ribbon, click on the *Generate* button 

to split the surface.
18. Name the fluid zone and the wall sections. In the top ribbon, *Tools* → *Named Selection*. Multiple faces/edges can be assigned to a single *Named Selection*. Click *Generate* 

after each assignment.

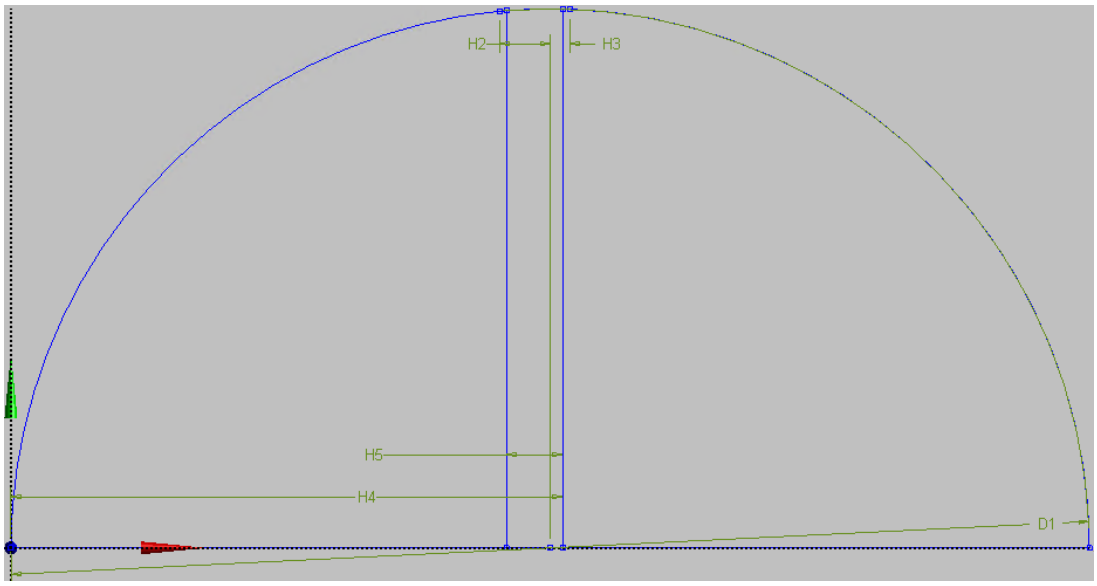


Figure 41 T16 tank geometry sketch.

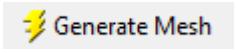
Table 16 T16 sketch dimensions.

Dimension Name	Value
D1	230 mm
H2	10.8 mm
H3	4.2 mm
H4	117.7 mm
H5	12 mm

Table 17 T16 named selections.

Named Selection	Number of Geometries	Faces/ Edges
fluid	3	Faces
axis	3	Edges
wall-dry	1	Edge
wall-adi	3	Edges
wall-wet	1	Edge
interior	2	Edges

Meshing

1. Drag and drop the *Mesh* module on top of the *Geometry* module to link the two.
2. Double click on *Mesh* to open the *Meshing* program.
3. In the *Outline*, select *Mesh*.
4. In the window, click the dropdown menu for *Solver Preference* and select *Fluent*.
5. Enter the intended cell size in *Element Size* (Pay attention to the units).
6. Generate the mesh by clicking on *Generate Mesh* 

in the top ribbon.
7. Export the mesh. In the top ribbon, *File* → *Export*. Name the file to your preference.

APPENDIX F COMMON FLUENT SETUP PROCEDURE FOR VALIDATION CASES

This appendix details a common setup procedure for both validation cases. Such common instructions include starting Fluent, importing the UDF, and creating report definitions. Case specific instructions will start from the last step of this appendix. For ease of use, the standalone Fluent program is used as opposed to the Workbench version.

1. Start *Fluent*.
    a. Under *Dimension*, select *2D*.
    b. Under *Processing Options*, choose either *Serial* or *Parallel*.
    c. If *Parallel* is chosen, choose a number of processes that is under the total number of logical processors in your system.
    d. If using *Fluent* in a *High-Performance Computing* unit, the number of logical processors available will depend on the number of processors assigned to your job.
    e. Under *Working Directory*, ensure this is the folder where you will read and write files for your simulation. The source codes and mesh file will need to be in this folder as well.
    f. Press *OK* to start *Fluent*.
2. Read the mesh. *File* → *Read* → *Mesh…*
3. In the *Tree, Setup* → *General* → *Solver*, select the *Transient* option under *Time* and select the *Axisymmetric* option under *2D Space*.
4. Compile the UDF library.
    a. In the ribbon, *User Defined* → *Functions* → *Compiled…* to open the *Compiled UDFs* dialog box.
    b. Under *Source Files*, select *Add…* and select the source codes, *Energy.c* and *Property.c* and click *OK*.
    c. Click on *Build* in the *Compiled UDFs* dialog box. If there are no errors, click on *Load*.
5. Create the UDS.
    a. In the ribbon, *User Defined* → *Scalars…* to open the *User-Defined Scalars* dialog box.
    b. Set the *Number of User-Defined Scalars* to 1.
    c. Set the *Flux Function* to the *uds_flux* and the *Unsteady Function* to *none*.
6. Create the UDMs.
    a. In the ribbon, *User Defined* → *Memory…* to open the *User-Defined Memory* dialog box.
    b. Set the number of *User-Defined Memory Locations* to 22.
7. Attach the function hooks.
    a. In the ribbon, *User Defined* → *Function Hooks…* to open the *User-Defined Function Hooks* dialog box.
    b. For *Initialization, Adjust,* and *Execute at End* press *Edit…* and add the respective function hooks.
8. Create the fluid for parahydrogen.
    a. In the *Tree, Setup* → *Materials* → *Fluid* → *air* → *Edit…*

b. For *Density, Viscosity,* and *UDS Diffusivity,* choose *user-defined* in the dropdown menu and select the respective property functions.
c. Press *Change/Create* to save the changes and close the *Create/Edit Materials* dialog box.

9. Setup the source terms for the simulation.
    a. In the *Tree, Setup → Cell Zone Conditions → fluid → Edit…*
    b. Select the *Source Terms* option and go to the *Source Terms* tab.
    c. Press *Edit…* to add source terms.
    d. For *Axial Momentum*, set *Number of Axial Momentum sources* to 2. Add the buoyancy term. Add the velocity suppression term *cancel_x*. Press *OK* to leave the dialog box.
    e. For *Radial Momentum*, set the *Number of Radial Momentum sources* to 1 and add the velocity suppression term *cancel_y*.
    f. For *User Scalar 0*, set *Number of User Scalar 0 sources* to 2 and select the sensible heat term and the latent heat term.
    g. Press *OK* to leave the dialog boxes and go back to the main window.
10. Modify spatial discretization settings.
    a. In the *Tree, Solution → Methods*. Double click to activate the *Task Page* for *Solution Methods*.
    b. For *Spatial Discretization,* select *PRESTO!* for *Pressure,* and select *First Order Upwind* for *Momentum*.
    c. The rest are left at default.
11. Modify the relaxation factors.
    a. In the *Tree, Solution → Controls*. Double click to activate the *Task Page* for *Solution Controls*.
    b. Set the *Under-Relaxation Factors* for *Pressure, Momentum,* and *User Scalar 0* to 0.1.
    c. The rest are left as default.
12. Create the *timestep-ender* report definition.
    a. In the *Tree, Solution → Report Definitions → New → User Defined…* to open the *User Defined Report Definition* dialog box.
    b. Name the report definition *timestep-ender* and choose *timestep_ender* from the *Function* dropdown menu.
13. Setup the timestep convergence criteria.
    a. In the *Tree, Solution → Monitors → Residual → Edit…* to open the *Residual Monitors* dialog box.
    b. Uncheck the *Print to Console* and *Plot* options under *Options*.
    c. Uncheck the *Check Convergence* options for all of the equations.
    d. Press *Convergence Conditions…* to open the *Convergence Conditions* dialog box.
    e. Select the *Time Step Convergence* option to add a convergence condition.
    f. Choose the *timestep-ender* in the *Report Definition* dropdown menu.
    g. Set the *Stop Criterion* to a very low value (1e-24) to prohibit unwanted activation.
    h. Press *OK* until back at the main window.
14. Create the rest of the report definitions.
    a. In the *Tree, Solution → Report Definitions → New → User Defined…* to open the *User Defined Report Definition* dialog box.

    b. Name the report definitions according to their names from the UDF.
15. Create report files to be output during the simulation.
    a. In the *Tree, Solution* → *Monitors* → *Report Files* → *New…* to create a new report file.
    b. Add all desired report definitions and press *Browse…* under *Output File Base Name*.
    c. Set the desired filename and include the .txt extension.
    d. Press *OK* until back to the main window.
16. Save the case. In the ribbon, *File* → *Write* → *Case…*
17. Further case-specific instructions are provided in their respective appendices.

APPENDIX G CONTINUED FLUENT SETUP FOR LOW GRAVITY CASE

This appendix details the case-specific setup of the case found in the Low Gravity Large Scale Orbital Fuel Tank section. It is a continuation of the general instructions presented in Appendix F. The source code for this case is provided in Appendix H.

1. Starting from the last item of Appendix F, assign the boundary conditions.
    a. In the *Tree, Setup → Boundary Conditions*.
    b. For *wall-fwd* and *wall-dry,* open the respective *Wall* dialog box and go to the *UDS* tab.
    c. Ensure the *User-Defined Scalar Boundary Condition* is set to *Specified Flux*.
    d. Set the *User-Defined Scalar Boundary Value* to the dry wall flux in the dropdown menu.
    e. The same procedure is done to *wall-aft* and *wall-wet* using the wet wall flux.
2. Initialize the solution.
    a. In the *Tree, Solution → Initialization → Initialize*.
    b. Ensure the starting liquid mass is close to 7257 kg.
3. Set the transient calculation options.
    a. In the *Tree, Solution → Run Calculation* to activate the *Run Calculation Task Page*.
    b. Choose a *Time Step Size* of 0.1 sec.
    c. Set the *Number of Time Steps* to 53600.
    d. Set *Max Iterations/Time Step* to at least 10000.
    e. Set a *Reporting Interval* to 10 or leave as 1.
4. Save the case again and set up autosaves and animations if desired.
5. Run the simulation.
    a. In the *Tree, Solution → Run Calculation*.
    b. Press *Calculate* to run.

APPENDIX H SOURCE CODE FOR LOW GRAVITY CASE

```
#include "udf.h"
#include "sg.h" /* This is needed for function BOUNDARY_FACE_GEOMETRY */

/* External Function Prototypes */
extern double meanof(double, double, double);
extern double prop(double, double, double, int);
extern double TsatfPsat(double, double);
extern void PsatfTsat(double, double*, double*);
extern double PfTR(double, double);
extern void RfTsat(double, double*, double*);

/* Pointer Variables */
double pk1, pk2, prho1, prho2, pe1, pe2, pel, pev, pmu1, pmu2, pcv1, pcv2;

/* Thread Indices */
#define fluid 7
#define i_fwd 9
#define i_dry 10
#define i_adi 11
#define i_wet 12
#define i_aft 13
#define i_com 14
#define i_start 9
#define i_stop 14

/* Wall Variables */
double A_wall[17];
#define A_fwd A_wall[i_fwd]
#define A_dry A_wall[i_dry]
#define A_adi A_wall[i_adi]
#define A_wet A_wall[i_wet]
#define A_aft A_wall[i_aft]
#define A_com A_wall[i_com]

/* Heat flux */
double HF_wall[17];
#define HF_fwd HF_wall[i_fwd]
#define HF_dry HF_wall[i_dry]
#define HF_adi HF_wall[i_adi]
#define HF_wet HF_wall[i_wet]
#define HF_aft HF_wall[i_aft]
#define HF_com HF_wall[i_com]

/* Heat rate */
double HR_wall[17];
#define HR_fwd HR_wall[i_fwd]
#define HR_dry HR_wall[i_dry]
#define HR_adi HR_wall[i_adi]
#define HR_wet HR_wall[i_wet]
#define HR_aft HR_wall[i_aft]
#define HR_com HR_wall[i_com]

/* Heat input */
double HI_wall[17];
#define HI_fwd HI_wall[i_fwd]
#define HI_dry HI_wall[i_dry]
#define HI_adi HI_wall[i_adi]
#define HI_wet HI_wall[i_wet]
#define HI_aft HI_wall[i_aft]
#define HI_com HI_wall[i_com]

/* Constants */
double top = 0.0;
double interf = 6.961;      /* axial location [m] of liq-vap interface; fill level should be 50.5% */
double P_init = 85495;      /* Initial Pressure */
double T_init = 19.71;      /* Initial Guessed Temperature */
double geom_mult = 2 * M_PI; /* Volume and area values are multiplied by this to find true volume/area */

/* User Defined Scalar */
#define T(c, t) C_UDSI(c, t, 0)
#define T_m1(c, t) C_UDSI_M1(c, t, 0)
#define FT(f, t) F_UDSI(f, t, 0)

/* User Defined Memory Variables */
#define eof(c, t) C_UDMI(c, t, 0)       /* Phase Fraction */
#define e(c, t) C_UDMI(c, t, 1)         /* Internal Energy [J/kg] */
#define e_sen(c, t) C_UDMI(c, t, 2)     /* Sensible Heat [J/kg] */
#define cv(c, t) C_UDMI(c, t, 3)        /* Specific Heat [J/kg-K] */
#define rho(c, t) C_UDMI(c, t, 4)       /* Density [kg/m3] */
#define k(c, t) C_UDMI(c, t, 5)         /* Thermal Conductivity [W/m-K] */
#define mu(c, t) C_UDMI(c, t, 6)        /* Viscosity [kg/m-s] */

/* Fluid properties that need to be stored for autosaves */
#define C_mass_tot(c, t) C_UDMI(c, t, 7)
#define C_mass_vap_t0(c, t) C_UDMI(c, t, 8)
#define C_mass_liq_t0(c, t) C_UDMI(c, t, 9)
#define C_P_sat(c, t) C_UDMI(c, t, 10)
#define C_T_sat(c, t) C_UDMI(c, t, 11)
#define C_delta_e(c, t) C_UDMI(c, t, 12)

/* Fluid properties from last time step */
#define eofold(c, t) C_UDMI(c, t, 13)
#define eof_m1(c, t) C_UDMI(c, t, 14)
#define e_m1(c, t) C_UDMI(c, t, 15)
#define e_sen_m1(c, t) C_UDMI(c, t, 16)
#define cv_m1(c, t) C_UDMI(c, t, 17)
#define rho_m1(c, t) C_UDMI(c, t, 18)

/* Extra UDMs */
#define C_interface(c, t) C_UDMI(c, t, 19)
#define C_search(c, t) C_UDMI(c, t, 20)
#define C_power(c, t) C_UDMI(c, t, 21)

/* For Axisymmetric Geometries */
#define vol(c, t) geom_mult*C_VOLUME(c, t)

/* Residuals */
double tol_mass = 1e-4, tol_vel = 1e-6, tol_uds0 = 1e-6, tol_eof = 1e-6, tol_p = 1e-6, tol_elf = 1e-6;
```

```
double res[7];
#define r_mass res[0]
#define r_xvel res[1]
#define r_yvel res[2]
#define r_temp res[3]
#define r_eof res[4]
#define r_pres res[5]
#define r_elf res[6]
double r_eof1;
double r_eof2;
char res_label[7][11] = {
        "continuity",
        "x-velocity",
        "y-velocity",
        "    energy",
        "       eof",
        "  pressure",
        "       elf"
};

/* Fluid properties shown in residuals */
double sat[8];
#define T_vap_bulk sat[0]       /* Bulk Vapor Temperature */
#define rho_vap_bulk sat[1]     /* Bulk Vapor Density */
#define P_sat sat[2]            /* Saturation Pressure */
#define T_sat sat[3]            /* Saturation Temperature */
#define elf sat[4]              /* Evaporated Liquid Fraction */
#define delta_e sat[5]          /* Latent Heat */
#define T_liq_bulk sat[6]/* Bulk Liquid Temperature */
#define rho_liq_bulk sat[7]     /* Bulk Liquid Density */
char sat_label[8][9] = {
        "T-V-Bulk",
        "R-V-Bulk",
        "Sat.Pres",
        "Sat.Temp",
        "EvapLiqF",
        "Lat.Heat",
        "T-L-Bulk",
        "R-L-Bulk"
};

/* Fluid Properties not shown in residuals */
double mass_tot;            /* Total Mass at Initialization (Stays Constant) */
double mass_liq;            /* Mass of Liquid Phase at Current Time */
double mass_liq_t0;             /* Mass of Liquid Phase at Initialization */
double mass_vap;            /* Mass of Vapor Phase at Current Time */
double mass_vap_t0;             /* Mass of Vapor Phase at Initialization */
double vol_tot;                 /* Total domain volume */
double fill;                /* Percent fill of tank */
double T_liq_sum, vol_liq_sum;      /* Bulk liquid properties */
double T_vap_sum, vol_vap_sum;      /* Bulk vapor properties */
double real_mass_tot, real_mass_vap, real_mass_liq;     /* Masses based on density (Vapor mass calculated
from vapor density) */
double max_liq_T, min_vap_T;/* Maximum temperatures found in fluid domain */
double gravity, rho0;

/* Iteration Variables and Parameters */
double resid;       /* temporary residual */
double sum;         /* temporary summation */
double eofsum;                  /* summation of mixed eofs for residual calculation */
double eofchi = 0.8;        /* eof relaxation factor */
double P_sat_chi = 0.3;   /* pressure relaxation factor */
int flag;       /* switch to trigger timestep convergence report definition */
int iter, res_count, count; /* iteration counters */
double dT = 0.1;    /* temperature interval used for energy derivative */
double interf_new, cell_size, x_start;
int col;

DEFINE_EXECUTE_ON_LOADING(execute_on_loading, libname)
{
        iter = 0;
#if !RP_HOST
        Domain *d = Get_Domain(1);
        Thread *t = Lookup_Thread(d, fluid);
        Thread *t_dry = Lookup_Thread(d, i_dry);
        Thread *t_wet = Lookup_Thread(d, i_wet);
        Thread *t_adi = Lookup_Thread(d, i_adi);
        Thread *t_fwd = Lookup_Thread(d, i_fwd);
        Thread *t_aft = Lookup_Thread(d, i_aft);
        Thread *t_com = Lookup_Thread(d, i_com);
        Thread *tt;
        cell_t c;
        face_t f;
        int i;

        /* Calculate Wall Areas */
        Message0("\nCalculating Wall Areas...\n");
        real Area[ND_ND], ds, es[ND_ND], A_by_es, dr0[ND_ND];
        for (i = i_start;i <= i_stop;i++) A_wall[i] = 0;
        begin_c_loop_int(c, t)
        {
                c_face_loop(c, t, i)
                {
                        f = C_FACE(c, t, i);
                        tt = C_FACE_THREAD(c, t, i);
                        if (BOUNDARY_FACE_THREAD_P(tt))
                        {
                                BOUNDARY_FACE_GEOMETRY(f, tt, Area, ds, es, A_by_es, dr0);
                                if (tt == t_dry) A_dry += NV_MAG(Area);
                                if (tt == t_wet) A_wet += NV_MAG(Area);
                                if (tt == t_adi) A_adi += NV_MAG(Area);
                                if (tt == t_fwd) A_fwd += NV_MAG(Area);
                                if (tt == t_aft) A_aft += NV_MAG(Area);
                                if (tt == t_com) A_com += NV_MAG(Area);
                        }
                }
        }
        end_c_loop_int(c, t)

        /* Sum all values across compute nodes */
        for (i = i_start;i <= i_stop;i++) A_wall[i] = geom_mult * PRF_GRSUM1(A_wall[i]);

        Message0("\nWall Areas:\n");
        Message0("Ullage Wall     %8.7g m2\n", A_dry);
        Message0("Liquid Wall     %8.7g m2\n", A_wet);
        Message0("Adiabatic Wall  %8.7g m2\n", A_adi);
```

```
	Message0("Forward Dome    %8.7g m2\n", A_fwd);
	Message0("Aft Dome        %8.7g m2\n", A_aft);
	Message0("Common Bulkhead %8.7g m2\n", A_com);
	Message0("Total Area for Walls Receiving Heat %8.7g m2\n", (A_dry + A_wet + A_fwd + A_aft));
	if (A_dry == 0 || A_wet == 0) Message0("\nERROR!\nThe wall areas must not be 0.\nActivate
'calculate_wall_areas' in Execute on Demand before proceeding.\n");
#endif
	node_to_host_real_6(A_dry, A_wet, A_adi, A_fwd, A_aft, A_com);
}

DEFINE_ON_DEMAND(calculate_wall_areas)
{
#if !RP_HOST
	Domain *d = Get_Domain(1);
	Thread *t = Lookup_Thread(d, fluid);
	Thread *t_dry = Lookup_Thread(d, i_dry);
	Thread *t_wet = Lookup_Thread(d, i_wet);
	Thread *t_adi = Lookup_Thread(d, i_adi);
	Thread *t_fwd = Lookup_Thread(d, i_fwd);
	Thread *t_aft = Lookup_Thread(d, i_aft);
	Thread *t_com = Lookup_Thread(d, i_com);
	Thread *tt;
	cell_t c;
	face_t f;
	int i;

	/* Calculate Wall Areas */
	Message0("\nCalculating Wall Areas...\n");
	real Area[ND_ND], ds, es[ND_ND], A_by_es, dr0[ND_ND];
	for (i = i_start;i <= i_stop;i++) A_wall[i] = 0;
	begin_c_loop_int(c, t)
	{
		c_face_loop(c, t, i)
		{
			f = C_FACE(c, t, i);
			tt = C_FACE_THREAD(c, t, i);
			if (BOUNDARY_FACE_THREAD_P(tt))
			{
				BOUNDARY_FACE_GEOMETRY(f, tt, Area, ds, es, A_by_es, dr0);
				if (tt == t_dry) A_dry += NV_MAG(Area);
				if (tt == t_wet) A_wet += NV_MAG(Area);
				if (tt == t_adi) A_adi += NV_MAG(Area);
				if (tt == t_fwd) A_fwd += NV_MAG(Area);
				if (tt == t_aft) A_aft += NV_MAG(Area);
				if (tt == t_com) A_com += NV_MAG(Area);
			}
		}
	}
	end_c_loop_int(c, t)

		/* Sum all values across compute nodes */
		for (i = i_start;i <= i_stop;i++) A_wall[i] = geom_mult * PRF_GRSUM1(A_wall[i]);

	Message0("\nWall Areas:\n");
	Message0("Ullage Wall     %8.7g m2\n", A_dry);
	Message0("Liquid Wall     %8.7g m2\n", A_wet);
	Message0("Adiabatic Wall  %8.7g m2\n", A_adi);
	Message0("Forward Dome    %8.7g m2\n", A_fwd);
	Message0("Aft Dome        %8.7g m2\n", A_aft);
	Message0("Common Bulkhead %8.7g m2\n", A_com);
	Message0("Total Area for Walls Receiving Heat %8.7g m2\n", (A_dry + A_wet + A_fwd + A_aft));
	if (A_dry == 0 || A_wet == 0) Message0("\nERROR!\nThe wall areas must not be 0.\nActivate
'calculate_wall_areas' in Execute on Demand before proceeding.\n");
#endif
	node_to_host_real_6(A_dry, A_wet, A_adi, A_fwd, A_aft, A_com);
}

DEFINE_INIT(hook_init, d)
{
#if !RP_HOST
	Message0("\nBeginning Initialization...\n");
	if (A_dry == 0 || A_wet == 0) Message0("Activate 'calculate_wall_areas' in Execute on Demand
before proceeding.\n");

	Thread *t = Lookup_Thread(d, fluid);
	Thread *t_dry = Lookup_Thread(d, i_dry);
	Thread *t_wet = Lookup_Thread(d, i_wet);
	Thread *t_adi = Lookup_Thread(d, i_adi);
	Thread *t_fwd = Lookup_Thread(d, i_fwd);
	Thread *t_aft = Lookup_Thread(d, i_aft);
	Thread *t_com = Lookup_Thread(d, i_com);
	cell_t c;
	face_t f;
	Node *node;
	int i;

	real x[ND_ND];

	/* Reset All Values */
	iter = 0;
	eofsum = 1;

	interf_new = interf;

	/* Patch EOF */
	begin_c_loop(c, t)
	{
		C_CENTROID(x, c, t);
		real max = -1e6;
		real min = 1e6;
		c_node_loop(c, t, i)
		{
			node = C_NODE(c, t, i);
			if (NODE_X(node) > max) max = NODE_X(node);
			if (NODE_X(node) < min) min = NODE_X(node);
		}
		if (max > interf && min > interf) eof(c, t) = 0;
		if (max > interf && min < interf) eof(c, t) = (interf - min) / (max - min);
		if (max < interf && min < interf) eof(c, t) = 1;
	}
	end_c_loop(c, t)

	/* Mark search zone */
	begin_c_loop(c, t)
	{
		C_search(c, t) = 0;
```

```
        C_CENTROID(x, c, t);
        if ((interf - 0.425 < x[0]) && (x[0] < interf + 0.175)) C_search(c, t) = 1;
}
end_c_loop(c, t)

/* Mark interface cells */
begin_c_loop(c, t)
{
        if ((0 < eof(c, t)) && (eof(c, t) < 1) && (C_search(c, t)))
        {
                C_interface(c, t) = 1;
                C_power(c, t) = 16;
        }
        else
        {
                C_interface(c, t) = 0;
                C_power(c, t) = -4;
        }
}
end_c_loop(c, t)

/* Calculate cell size */
sum = 0;
count = 0;
Message0("\nCalculating Cell Sizes...\n");
begin_c_loop_int(c, t)
{
        if (C_search(c, t))
        {
                C_CENTROID(x, c, t);
                sum += vol(c, t) / (2 * M_PI*x[1]);
                count++;
        }
}
end_c_loop_int(c, t)
sum = PRF_GRSUM1(sum);
count = PRF_GISUM1(count);
cell_size = sqrt(sum / count);
Message0(" Average size of %d cells: %10.4e m\n", count, cell_size);
cell_size = 1e-2*round(cell_size / 1e-2);
Message0(" Rounded Cell Size: %10.4e m\n", cell_size);

col = round(0.6 / cell_size);
Message0(" %d Columns in search zone\n", col);
x_start = interf - 0.425;

/* Calculate Saturation Properties */
P_sat = P_init;
T_init = TsatfPsat(P_init, T_init);
T_sat = T_init;
RfTsat(T_sat, &prho1, &prho2);
pel = prop(T_sat, prho1, P_sat, 2);
pev = prop(T_sat, prho2, P_sat, 2);
delta_e = pev - pel;

/* Patch Temperatures */
begin_c_loop(c, t)
{
        C_CENTROID(x, c, t);
        if (x[0] >= interf) T(c, t) = T_sat;
        else T(c, t) = (25 / 9) / (top - interf)*(x[0] - interf) + T_sat;
}
end_c_loop(c, t)
begin_f_loop(f, t_dry)
{
        F_CENTROID(x, f, t_dry);
        FT(f, t_dry) = (25 / 9) / (top - interf)*(x[0] - interf) + T_sat;
}
end_f_loop(f, t_dry)
begin_f_loop(f, t_wet)
{
        FT(f, t_wet) = T_sat;
}
end_f_loop(f, t_wet)
begin_f_loop(f, t_adi)
{
        F_CENTROID(x, f, t_adi);
        if (x[0] >= interf) FT(f, t_adi) = T_sat;
        else FT(f, t_adi) = (25 / 9) / (top - interf)*(x[0] - interf) + T_sat;
}
end_f_loop(f, t_adi)
begin_f_loop(f, t_fwd)
{
        F_CENTROID(x, f, t_fwd);
        FT(f, t_fwd) = (25 / 9) / (top - interf)*(x[0] - interf) + T_sat;
}
end_f_loop(f, t_fwd)
begin_f_loop(f, t_aft)
{
        FT(f, t_aft) = T_sat;
}
end_f_loop(f, t_aft)
begin_f_loop(f, t_com)
{
        FT(f, t_com) = T_sat;
}
end_f_loop(f, t_com)

/* Initialize Flow Properties */
begin_c_loop(c, t)
{
        /* Liquid */
        if (eof(c, t) <= 0)
        {
                rho(c, t) = prop(T(c, t), 70.0, P_sat, 1);
                e(c, t) = prop(T(c, t), rho(c, t), P_sat, 2);
                cv(c, t) = prop(T(c, t), rho(c, t), P_sat, 3);
                k(c, t) = prop(T(c, t), rho(c, t), P_sat, 4);
                mu(c, t) = prop(T(c, t), rho(c, t), P_sat, 5);
        }
        /* Mixed Phase */
        if ((eof(c, t) > 0) && (eof(c, t) < 1))
        {
                RfTsat(T(c, t), &prho1, &prho2);
                rho(c, t) = meanof(prho1, prho2, eof(c, t));
```

```
                pe1 = prop(T(c, t), prho1, P_sat, 2);
                pe2 = prop(T(c, t), prho2, P_sat, 2);
                e(c, t) = meanof(pe1, pe2, eof(c, t));

                pcv1 = prop(T(c, t), prho1, P_sat, 3);
                pcv2 = prop(T(c, t), prho2, P_sat, 3);
                cv(c, t) = meanof(pcv1, pcv2, eof(c, t));

                pk1 = prop(T(c, t), prho1, P_sat, 4);
                pk2 = prop(T(c, t), prho2, P_sat, 4);
                k(c, t) = meanof(pk1, pk2, eof(c, t));

                pmu1 = prop(T(c, t), prho1, P_sat, 5);
                pmu2 = prop(T(c, t), prho2, P_sat, 5);
                mu(c, t) = meanof(pmu1, pmu2, eof(c, t));
        }
        /* Vapor */
        if (eof(c, t) >= 1)
        {
                rho(c, t) = prop(T(c, t), 1.0, P_sat, 1);
                e(c, t) = prop(T(c, t), rho(c, t), P_sat, 2);
                cv(c, t) = prop(T(c, t), rho(c, t), P_sat, 3);
                k(c, t) = prop(T(c, t), rho(c, t), P_sat, 4);
                mu(c, t) = prop(T(c, t), rho(c, t), P_sat, 5);
        }
        /* Sensible Heat */
        e_sen(c, t) = e(c, t) - eof(c, t)*delta_e;

        /* Previous timestep values */
        eofold(c, t) = eof(c, t);
        eof_m1(c, t) = eof(c, t);
        e_m1(c, t) = e(c, t);
        e_sen_m1(c, t) = e_sen(c, t);
        cv_m1(c, t) = cv(c, t);
        rho_m1(c, t) = rho(c, t);
    }
    end_c_loop(c, t)

    /* Calculate System Mass and Bulk Properties */
    mass_tot = 0, vol_tot = 0, mass_liq_t0 = 0, T_liq_sum = 0, vol_liq_sum = 0, T_vap_sum = 0,
vol_vap_sum = 0;
    begin_c_loop_int(c, t) /* Interior cell loop prevents double counting of cells between domain
partitions */
    {
        mass_tot += rho(c, t) * vol(c, t);
        vol_tot += vol(c, t);
        if (eof(c, t) != 1)
        {
            mass_liq_t0 += prop(T(c, t), 70.0, P_sat, 1) * vol(c, t)*(1 - eof(c, t));
            T_liq_sum += T(c, t)*vol(c, t)*(1 - eof(c, t));
            vol_liq_sum += vol(c, t)*(1 - eof(c, t));
        }
        if (eof(c, t) != 0)
        {
            T_vap_sum += T(c, t)*vol(c, t)*eof(c, t);
            vol_vap_sum += vol(c, t)*eof(c, t);
        }
    }
    end_c_loop_int(c, t)
    /* PRF_GRSUM performs summations across all processing nodes */
    mass_tot = PRF_GRSUM1(mass_tot);
    mass_liq_t0 = PRF_GRSUM1(mass_liq_t0);
    mass_vap_t0 = mass_tot - mass_liq_t0;
    vol_tot = PRF_GRSUM1(vol_tot);
    /* Calculate bulk properties */
    mass_liq = mass_liq_t0;
    T_liq_sum = PRF_GRSUM1(T_liq_sum);
    vol_liq_sum = PRF_GRSUM1(vol_liq_sum);
    T_liq_bulk = T_liq_sum / vol_liq_sum;
    rho_liq_bulk = mass_liq_t0 / vol_liq_sum;
    mass_vap = mass_vap_t0;
    T_vap_sum = PRF_GRSUM1(T_vap_sum);
    vol_vap_sum = PRF_GRSUM1(vol_vap_sum);
    T_vap_bulk = T_vap_sum / vol_vap_sum;
    rho_vap_bulk = mass_vap_t0 / vol_vap_sum;
    fill = 100 * vol_liq_sum / vol_tot;

    /* Export to UDM */
    begin_c_loop(c, t)
    {
        C_P_sat(c, t) = P_sat;
        C_T_sat(c, t) = T_sat;
        C_delta_e(c, t) = delta_e;
        C_mass_tot(c, t) = mass_tot;
        C_mass_vap_t0(c, t) = mass_vap_t0;
        C_mass_liq_t0(c, t) = mass_liq_t0;
    }
    end_c_loop(c, t)

    /* Calculate real masses */
    real_mass_tot = 0, real_mass_vap = 0, real_mass_liq = 0;
    begin_c_loop_int(c, t)
    {
        real_mass_tot += rho(c, t)*vol(c, t);
        if (eof(c, t) != 0) real_mass_vap += prop(T(c, t), 1.0, P_sat, 1)*vol(c, t)*eof(c, t);
        if (eof(c, t) != 1) real_mass_liq += prop(T(c, t), 70.0, P_sat, 1)*vol(c, t)*(1 - eof(c, t));
    }
    end_c_loop_int(c, t)
    real_mass_tot = PRF_GRSUM1(real_mass_tot);
    real_mass_vap = PRF_GRSUM1(real_mass_vap);
    real_mass_liq = PRF_GRSUM1(real_mass_liq);
    rho0 = real_mass_vap / vol_vap_sum;

    /* Display Initialized Data */
    Message0("\nInitialized Flow Variables:\n\n");
    Message0(" Total Mass  Liquid Mass  Vapor Mass\n");
    Message0(" %#10.4e   %#10.4e  %#10.4e\n\n", mass_tot, mass_liq_t0, mass_vap_t0);
    Message0(" Vapor Vol.  T-V-Bulk  R-V-Bulk ");
    Message0(" LiquidVol.  T-L-Bulk  R-L-Bulk\n");
    Message0(" %#10.4e  %#8.7g  %#8.7g ", vol_vap_sum, T_vap_bulk, rho_vap_bulk);
    Message0(" %#10.4e  %#8.7g  %#8.7g\n\n", vol_liq_sum, T_liq_bulk, rho_liq_bulk);
    Message0(" Sat.Pres  Sat.Temp  Lat.Heat\n");
    Message0(" %#8.7g  %#8.7g  %#8.7g\n", P_sat, T_sat, delta_e);
    Message0("\nPercent Fill: %#4.3g%%\n", fill);
    Message0("\nInitialization Complete!\n");
#endif
```

```
}

/**************************
Start FLUENT Iteration Loop
**************************/

DEFINE_PROFILE(profile_flux_dry_wall, t, i)
{
	face_t f;
	real time = CURRENT_TIME;
	HR_dry = (-1.36820e-17)*pow(time, 6) + (1.91965e-13)*pow(time, 5) - (1.01026e-9)*pow(time, 4) +
(2.52341e-6)*pow(time, 3) - (2.00413e-3)*pow(time, 2) - (4.95785)*time + (1.61470e+4);
	HF_dry = HR_dry / (A_dry + A_fwd);
	HF_fwd = HF_dry;
	begin_f_loop(f, t)
	{
		F_PROFILE(f, t, i) = HF_dry;
	}
	end_f_loop(f, t)
}
DEFINE_PROFILE(profile_flux_wet_wall, t, i)
{
	face_t f;
	real time = CURRENT_TIME;
	HR_wet = (-6.16135e-17)*pow(time, 6) + (9.11402e-13)*pow(time, 5) - (5.03916e-9)*pow(time, 4) +
(1.27767e-5)*pow(time, 3) - (1.25141e-2)*pow(time, 2) - (3.76164)*time + (2.14191e+4);
	HF_wet = HR_wet / (A_wet + A_aft);
	HF_aft = HF_wet;
	begin_f_loop(f, t)
	{
		F_PROFILE(f, t, i) = HF_wet;
	}
	end_f_loop(f, t)
}

DEFINE_ADJUST(hook_adjust, d)
{
#if !RP_HOST
	Thread *t = Lookup_Thread(d, fluid);
	cell_t c;
	int i;

	if (iter == 0)
	{
		r_eof = 0;
		r_eof1 = 0;
		r_eof2 = 0;
		r_pres = 0;
		r_elf = 0;
		res_count = 0;
	}

	int iter_old = iter;
	iter++;
	flag = 0;
	/* Obtain residual */
	if (iter > 1)
	{
		for (i = 0; i <= 3; i++)
		{
			if (DOMAIN_RES_SCALE(d, i) == 0) res[i] = DOMAIN_RES(d, i);
			else res[i] = DOMAIN_RES(d, i) / DOMAIN_RES_SCALE(d, i);
		}
}

/* Import from UDM */
begin_c_loop(c, t)
{
	mass_tot = C_mass_tot(c, t);
	mass_vap_t0 = C_mass_vap_t0(c, t);
	mass_liq_t0 = C_mass_liq_t0(c, t);
	P_sat = C_P_sat(c, t);
	T_sat = C_T_sat(c, t);
	delta_e = C_delta_e(c, t);
}
end_c_loop(c, t)

PRF_GSYNC();
/* Calculate Mass and Bulk Vapor Properties */
mass_liq = 0, T_liq_sum = 0, vol_liq_sum = 0, mass_vap = 0, T_vap_sum = 0, vol_vap_sum = 0;
begin_c_loop_int(c, t)
{
	if (eof(c, t) != 1)
	{
		mass_liq += prop(T(c, t), 70.0, P_sat, 1) * vol(c, t)*(1 - eof(c, t));
		T_liq_sum += T(c, t)*vol(c, t)*(1 - eof(c, t));
		vol_liq_sum += vol(c, t)*(1 - eof(c, t));
	}
	if (eof(c, t) != 0);
	{
		T_vap_sum += T(c, t)*vol(c, t)*eof(c, t);
		vol_vap_sum += vol(c, t)*eof(c, t);
	}
}
end_c_loop_int(c, t)
mass_liq = PRF_GRSUM1(mass_liq);
T_liq_sum = PRF_GRSUM1(T_liq_sum);
vol_liq_sum = PRF_GRSUM1(vol_liq_sum);
T_liq_bulk = T_liq_sum / vol_liq_sum;
rho_liq_bulk = mass_liq / vol_liq_sum;
mass_vap = mass_tot - mass_liq;
T_vap_sum = PRF_GRSUM1(T_vap_sum);
vol_vap_sum = PRF_GRSUM1(vol_vap_sum);
T_vap_bulk = T_vap_sum / vol_vap_sum;
rho_vap_bulk = mass_vap / vol_vap_sum;
fill = 100 * vol_liq_sum / vol_tot;

/* Obtain real masses */
PRF_GSYNC();
real_mass_tot = 0, real_mass_vap = 0, real_mass_liq = 0;
begin_c_loop_int(c, t)
{
	real_mass_tot += rho(c, t)*vol(c, t);
	if (eof(c, t) != 0) real_mass_vap += prop(T(c, t), 1.0, P_sat, 1)*vol(c, t)*eof(c, t);
	if (eof(c, t) != 1) real_mass_liq += prop(T(c, t), 70.0, P_sat, 1)*vol(c, t)*(1 - eof(c, t));
}
```

```
end_c_loop_int(c, t)
real_mass_tot = PRF_GRSUM1(real_mass_tot);
real_mass_vap = PRF_GRSUM1(real_mass_vap);
real_mass_liq = PRF_GRSUM1(real_mass_liq);
rho0 = real_mass_vap / vol_vap_sum;

/* Calculate saturation pressure */
real P_sat_old = P_sat; /* Store pressure for residual */
P_sat = PfTR(T_vap_bulk, rho_vap_bulk); /* Update pressure using bulk vapor properties */
r_pres = fabs(P_sat - P_sat_old) / P_sat_old; /* Calculate pressure residual */
P_sat = (P_sat_chi)*P_sat + (1 - P_sat_chi)*P_sat_old; /* Relax pressure */

/* Calculate saturation temperature */
T_sat = TsatfPsat(P_sat, T_sat);

/* Calculate saturation energies and latent heat */
RfTsat(T_sat, &prho1, &prho2);
pel = prop(T_sat, prho1, P_sat, 2);
pev = prop(T_sat, prho2, P_sat, 2);
delta_e = pev - pel;

/* Calculate evaporated liquid fraction */
real elf_old = elf;
elf = 1 - (mass_liq / mass_liq_t0);
r_elf = fabs((elf - elf_old) / elf_old);

/* Calculate Density, Internal Energy, and EOF */
PRF_GSYNC();
begin_c_loop(c, t)
{
	/* Determine density and energy using new temperature and old eof */
	if (eof(c, t) == 0)
	{
		rho(c, t) = prop(T(c, t), rho(c, t), P_sat, 1);
		e(c, t) = prop(T(c, t), rho(c, t), P_sat, 2);
	}
	if (eof(c, t) > 0 && eof(c, t) < 1)
	{
		RfTsat(T(c, t), &prho1, &prho2);
		rho(c, t) = meanof(prho1, prho2, eof(c, t));

		pe1 = prop(T(c, t), prho1, P_sat, 2);
		pe2 = prop(T(c, t), prho2, P_sat, 2);
		e(c, t) = meanof(pe1, pe2, eof(c, t));
	}
	if (eof(c, t) == 1)
	{
		rho(c, t) = prop(T(c, t), rho(c, t), P_sat, 1);
		e(c, t) = prop(T(c, t), rho(c, t), P_sat, 2);
	}
	e_sen(c, t) = e(c, t) - eof(c, t)*delta_e;

	/* Save old eof to calculate residual */
	eofold(c, t) = eof(c, t);

	/* Determine new eof using energy */
	if (e(c, t) > pel && e(c, t) < pev) eof(c, t) = (e(c, t) - pel) / delta_e;
	if (e(c, t) <= pel) eof(c, t) = 0;
	if (e(c, t) >= pev) eof(c, t) = 1;
}
end_c_loop(c, t)

/* Calculate eof residual */
PRF_GSYNC();
real eofsumold = eofsum;
resid = 0, r_eof1 = 0, r_eof2 = 0, eofsum = 1;
begin_c_loop_int(c, t)
{
	/* Method 1: Maximum Cell Change */
	if (eofold(c, t) <= 0) resid = eof(c, t);
	else resid = fabs(eofold(c, t) - eof(c, t) / eofold(c, t));
	if (resid > r_eof1) r_eof1 = resid;

	/* Method 2: Change in the Sum of Mushy Zone */
	if (eof(c, t) > 0 && eof(c, t) < 1) eofsum += eof(c, t);
}
end_c_loop_int(c, t)
r_eof1 = PRF_GRHIGH1(r_eof1);
eofsum = PRF_GRSUM1(eofsum);
r_eof2 = fabs(eofsum - eofsumold) / eofsumold;
r_eof = r_eof2; /* Choose method for residual */

/* Update properties from new eof and find max temps */
max_liq_T = 0.0, min_vap_T = 1000.0;
begin_c_loop(c, t)
{
	/* Relax EOF */
	eof(c, t) = eofchi * eof(c, t) + (1 - eofchi)*eofold(c, t);

	/* Liquid */
	if (eof(c, t) == 0)
	{
		cv(c, t) = prop(T(c, t), rho(c, t), P_sat, 3);
		k(c, t) = prop(T(c, t), rho(c, t), P_sat, 4);
		mu(c, t) = prop(T(c, t), rho(c, t), P_sat, 5);
	}
	/* Mixed Phase */
	if ((eof(c, t) > 0) && (eof(c, t) < 1))
	{
		RfTsat(T(c, t), &prho1, &prho2);

		pcv1 = prop(T(c, t), prho1, P_sat, 3);
		pcv2 = prop(T(c, t), prho2, P_sat, 3);
		cv(c, t) = meanof(pcv1, pcv2, eof(c, t));

		pk1 = prop(T(c, t), prho1, P_sat, 4);
		pk2 = prop(T(c, t), prho2, P_sat, 4);
		k(c, t) = meanof(pk1, pk2, eof(c, t));

		pmu1 = prop(T(c, t), prho1, P_sat, 5);
		pmu2 = prop(T(c, t), prho2, P_sat, 5);
		mu(c, t) = meanof(pmu1, pmu2, eof(c, t));
	}
	/* Vapor */
	if (eof(c, t) == 1)
	{
```

```
            cv(c, t) = prop(T(c, t), rho(c, t), P_sat, 3);
            k(c, t) = prop(T(c, t), rho(c, t), P_sat, 4);
            mu(c, t) = prop(T(c, t), rho(c, t), P_sat, 5);
        }

        /* Find min/max temperatures */
        if (eof(c, t) == 0 && T(c, t) > max_liq_T) max_liq_T = T(c, t);
        if (eof(c, t) == 1 && T(c, t) < min_vap_T) min_vap_T = T(c, t);
}
end_c_loop(c, t)
max_liq_T = PRF_GRHIGH1(max_liq_T);
min_vap_T = PRF_GRLOW1(min_vap_T);

/* Check convergence when pressure is below tolerance */
if (r_pres < tol_p) flag = 1;
if (r_elf > tol_elf) flag = 0;
if (r_eof > tol_eof) flag = 0;
if (r_temp > tol_uds0) flag = 0;
if (r_mass > tol_mass) flag = 0;
if (r_xvel > tol_vel) flag = 0;
if (r_yvel > tol_vel) flag = 0;
/* If the flag remains active after checking residuals,
then the report definition timestep_ender will trigger the end of the timestep for Fluent */

/* Export to UDM */
begin_c_loop(c, t)
{
        C_P_sat(c, t) = P_sat;
        C_T_sat(c, t) = T_sat;
        C_delta_e(c, t) = delta_e;
}
end_c_loop(c, t)

/* Display residual every reporting interval */
PRF_GSYNC();
if (iter%RP_Get_Integer("iteration-chunk") == 0)
{
        res_count++;
        if (res_count % 20 == 1)
        {
                Message0("\nTime = %g sec\n", CURRENT_TIME);
                Message0("  iter ");
                for (i = 0;i <= 6;i++) Message0(" %s ", res_label[i]);
                Message0("|");
                for (i = 0;i <= 3;i++) Message0(" %s ", sat_label[i]);
                Message0("  %s ", sat_label[4]); /* This one needs more space */
                Message0("  T-V-Min   T-L-Max ");
                Message0("\n");
        }
        Message0("%6d ", iter);
        for (i = 0;i <= 6;i++) Message0(" %10.4e ", res[i]);
        Message0("|");
        for (i = 0;i <= 3;i++) Message0(" %#8.7g ", sat[i]);
        Message0(" %+9.2e ", sat[4]);
        Message0(" %#8.7g  %#8.7g ", min_vap_T, max_liq_T);
        Message0("\n");
}
        /* If flag says timestep has converged, then make iter value constant to end timestep */
        if (flag == 1 && iter != 1 && iter%RP_Get_Integer("iteration-chunk") == 0)
        {
                Message0("Solution converged in %d iterations\n", iter);
                for (i = 0;i <= 6;i++) Message0(" %s ", res_label[i]);
                Message0("\n");
                for (i = 0;i <= 6;i++) Message0(" %10.4e ", res[i]);
                Message0("\n");
                iter = iter_old;
        }
#endif
}

DEFINE_UDS_FLUX(uds_flux, f, t, i)
{
        cell_t c0, c1 = -1;
        Thread *t0, *t1 = NULL;

        real  flux = 0.0;
        real Area[ND_ND], ds, es[ND_ND], A_by_es, dr0[ND_ND], dr1[ND_ND];
        c0 = F_C0(f, t);
        t0 = F_C0_THREAD(f, t);

        /* If face lies at domain boundary, use face values; */
        /* If face lies IN the domain, use average of adjacent cells. */
        if (BOUNDARY_FACE_THREAD_P(t)) /*Most face values will be available*/
        {
                flux = F_FLUX(f, t) * cv(c0, t0); /* flux through Face */
        }
        else
        {
                c1 = F_C1(f, t);   /* Get cell on other side of face */
                t1 = F_C1_THREAD(f, t);
                flux = F_FLUX(f, t) * (cv(c0, t0) + cv(c1, t1)) / 2.0; /* Average flux through face */
        }

        return flux;
}

DEFINE_SOURCE(source_sensible_heat, c, t, dS, eqn)
{
        real dt, source;
        dt = CURRENT_TIMESTEP;
        source = -rho_m1(c, t)*(e_sen(c, t) - e_sen_m1(c, t)) / dt;
        dS[eqn] = -rho_m1(c, t) * cv_m1(c, t) / dt;
        return source;
}
DEFINE_SOURCE(source_latent_heat, c, t, dS, eqn)
{
        real dt, source;
        dt = CURRENT_TIMESTEP;
        source = rho_m1(c, t) * delta_e * (eof_m1(c, t) - eof(c, t)) / dt;
        dS[eqn] = 0;
        return source;
}

DEFINE_SOURCE(source_bouyancy, c, t, dS, eqn)
{
```

```
        real source;
        real time = CURRENT_TIME;
        gravity = ((3.21349e-21)*pow(time, 6) - (5.81560e-17)*pow(time, 5) + (4.12226e-13)*pow(time, 4) -
(1.45614e-9)*pow(time, 3) + (2.76153e-6)*pow(time, 2) - (3.17385e-3)*time + (3.53157))*1e-4;
        source = gravity * 9.81 * (rho(c, t) - rho0);
        dS[eqn] = 0;
        return source;
}
DEFINE_SOURCE(cancel_x, c, t, dS, eqn)
{
        real source;
        if (C_interface(c, t))
        {
                real power = pow(10, C_power(c, t));
                source = -power * C_U(c, t);
                dS[eqn] = -power;
        }
        else
        {
                source = 0;
                dS[eqn] = 0;
        }
        return source;
}
DEFINE_SOURCE(cancel_y, c, t, dS, eqn)
{
        real source;
        if (C_interface(c, t))
        {
                real power = pow(10, C_power(c, t));
                source = -power * C_V(c, t);
                dS[eqn] = -power;
        }
        else
        {
                source = 0;
                dS[eqn] = 0;
        }
        return source;
}

DEFINE_PROPERTY(property_density, c, t)
{
        return 71.6553;
}
DEFINE_PROPERTY(property_viscosity, c, t)
{
        return mu(c, t);
}
DEFINE_DIFFUSIVITY(property_diffusivity, c, t, i)
{
        return k(c, t);
}

/*********************************************
End FLUENT Iteration Loop (Covergence Check)
*********************************************/
```

```
DEFINE_EXECUTE_AT_END(hook_execute_at_end)
{
        iter = 0;
        int i, ii;
#if !RP_HOST
        if (CURRENT_TIME > 0)
        {
                for (i = i_start;i <= i_stop; i++)
                {
                        HR_wall[i] = HF_wall[i] * A_wall[i];
                        HI_wall[i] += HR_wall[i] * CURRENT_TIMESTEP;
                }
        }
        Domain *d = Get_Domain(1);
        Thread *t = Lookup_Thread(d, fluid);
        cell_t c;
        Node *node;
        real x[ND_ND];

        /* Reconstruct Interface */
        if (N_TIME % 1 == 0)
        {
                real y_test, y_old, y_new, dVdy, dy = 0.00000001;
                int j;

                y_old = interf_new;
                Message0("\nOld Interface: %#7.6g\n", interf_new);

                int iter = 1;
                real tol = 1e-9;
                resid = 1.0;
                Message0(" i      y_test    vol_test       dVdy       y_old       y_new       resid\n");
                while (resid > tol && iter < 20)
                {
                        Message0(" %d ", iter);
                        double vol_test[3] = { 0 };
                        for (j = 0; j <= 2; j++)
                        {
                                y_test = y_old + (j - 1)*dy;
                                Message0(" %#10.8g ", y_test);

                                /* Recalculate EOF */
                                begin_c_loop(c, t)
                                {
                                        C_CENTROID(x, c, t);
                                        real max = -1e6;
                                        real min = 1e6;
                                        c_node_loop(c, t, i)
                                        {
                                                node = C_NODE(c, t, i);
                                                if (NODE_X(node) > max) max = NODE_X(node);
                                                if (NODE_X(node) < min) min = NODE_X(node);
                                        }
                                        if (max > y_test && min > y_test) eofold(c, t) = 0;
                                        if (max > y_test && min < y_test) eofold(c, t) = (y_test - min) / (max - min);
                                        if (max < y_test && min < y_test) eofold(c, t) = 1;
                                }
                                end_c_loop(c, t)
```

```
                /* Calculate new liquid volume */
                begin_c_loop_int(c, t)
                {
                        if (eofold(c, t) != 1) vol_test[j] += vol(c, t)*(1 - eofold(c, t));
                }
                end_c_loop_int(c, t)

                vol_test[j] = PRF_GRSUM1(vol_test[j]);
                Message0(" %10.4e ", vol_test[j]);
                if (j < 2) Message0("\n   ");
        }

        dVdy = (vol_test[2] - vol_test[0]) / (2 * dy);
        Message0(" %+10.3e ", dVdy);
        y_new = y_old - (vol_test[1] - vol_liq_sum) / dVdy;
        Message0(" %#10.8g  %#10.8g ", y_old, y_new);

        resid = fabs(y_new - y_old) / y_old;
        Message0(" %10.4e \n", resid);
        y_old = y_new;
        iter++;
    }
    if (resid > tol) Message0("Interface reconstruction did not converge!\n");
    interf_new = y_new;
    Message0("New Interface: %#7.6g  %10.4e\n", interf_new, resid);

    /* Reconstruct Interface */
    begin_c_loop(c, t)
    {
        C_CENTROID(x, c, t);
        real max = -1e6;
        real min = 1e6;
        c_node_loop(c, t, i)
        {
            node = C_NODE(c, t, i);
            if (NODE_X(node) > max) max = NODE_X(node);
            if (NODE_X(node) < min) min = NODE_X(node);
        }
        if (max > y_test && min > y_test) eof(c, t) = 0;
        if (max > y_test && min < y_test) eof(c, t) = (interf_new - min) / (max - min);
        if (max < y_test && min < y_test) eof(c, t) = 1;
    }
    end_c_loop(c, t)
}

/* Update source power */
if (N_TIME % 1 == 0)
{
    double* dfdtsum = (double*)calloc(col, sizeof(double));
    double* eofsum = (double*)calloc(col, sizeof(double));
    double* eofhigh = (double*)calloc(col, sizeof(double));
    double* eoflow = (double*)calloc(col, sizeof(double));
    for (i = 0;i < col;i++) eoflow[i] = 2;
    int* count_total = (int*)calloc(col, sizeof(int));
    double* pwrold = (double*)calloc(col, sizeof(double));
    double* pwrnew = (double*)calloc(col, sizeof(double));
    double* centroid = (double*)calloc(col, sizeof(double));
    real dt = CURRENT_TIMESTEP;

    /* Sum interface cells by column */
    begin_c_loop_int(c, t)
    {
        if (C_search(c, t))
        {
            /* Index column */
            C_CENTROID(x, c, t);
            for (i = 0;i < col;i++)
                if (x[0] < x_start + cell_size * (i + 1))
                {
                    ii = i;
                    break;
                }

            /* Sum column phase change data */
            dfdtsum[ii] += (eof(c, t) - eof_m1(c, t)) / dt;
            eofsum[ii] += eof(c, t);
            if (eof(c, t) > eofhigh[ii]) eofhigh[ii] = eof(c, t);
            if (eof(c, t) < eoflow[ii]) eoflow[ii] = eof(c, t);
            pwrold[ii] += C_power(c, t);
            centroid[ii] += x[0];
            count_total[ii]++;
        }
    }
    end_c_loop_int(c, t)

    /* Sum or sync across processing nodes */
    for (i = 0;i < col; i++)
    {
        dfdtsum[i] = PRF_GRSUM1(dfdtsum[i]);
        eofsum[i] = PRF_GRSUM1(eofsum[i]);
        eofhigh[i] = PRF_GRHIGH1(eofhigh[i]);
        eoflow[i] = PRF_GRLOW1(eoflow[i]);
        pwrold[i] = PRF_GRSUM1(pwrold[i]);
        centroid[i] = PRF_GRSUM1(centroid[i]);
        count_total[i] = PRF_GISUM1(count_total[i]);
    }

    /* Use column averages to determine where to put power */
    begin_c_loop(c, t)
    {
        if (C_search(c, t))
        {
            /* Index column */
            C_CENTROID(x, c, t);
            for (i = 0;i < col;i++)
                if (x[0] < x_start + cell_size * (i + 1))
                {
                    ii = i;
                    break;
                }

            /* Give power to cells when adjacent column is about to completely
condense/evaporate */
            real buffer = 0.15 - 2 * cell_size;
            if (ii < col)
```

```
                        if (dfdtsum[ii + 1] < 0) /* condensation */
                                if (eofsum[ii + 1] / count_total[ii + 1] < buffer)
                                        C_power(c, t) += dt * (1 + 10);
                        if (ii >= 1)
                                if (dfdtsum[ii - 1] > 0) /* evaporation */
                                        if (eofsum[ii - 1] / count_total[ii - 1] > 1.0 - buffer)
                                                C_power(c, t) += dt * (1 + 10);

                        /* Give power to mixed phase cells */
                        if (0 < eofsum[ii] / count_total[ii] && eofsum[ii] / count_total[ii] < 1) C_power(c,
t) += 1 * dt;

                        /* Remove power from pure phase cells */
                        if (eofhigh[ii] == 0 || eoflow[ii] == 1) C_power(c, t) -= 1 * dt;

                        /* Give power to cells when phase front is directly in between two columns */
                        if (ii < col)
                                if (eofhigh[ii + 1] == 0 && eoflow[ii] == 1)
                                        C_power(c, t) += 1 * dt;

                        /* enforce limits */
                        if (C_power(c, t) < -4) C_power(c, t) = -4;
                        if (C_power(c, t) > 16) C_power(c, t) = 16;

                        /* update interface mark */
                        if (C_power(c, t) > -4) C_interface(c, t) = 1;
                        else C_interface(c, t) = 0;
                }
        }
        end_c_loop(c, t)

        /* Calculate change for monitoring purposes */
        PRF_GSYNC();
        begin_c_loop_int(c, t)
        {
                if (C_search(c, t))
                {
                        C_CENTROID(x, c, t);
                        for (i = 0;i < col;i++)
                                if (x[0] < x_start + cell_size * (i + 1))
                                {
                                        ii = i;
                                        break;
                                }
                        pwrnew[ii] += C_power(c, t);
                }
        }
        end_c_loop_int(c, t)
        for (i = 0;i < col; i++) pwrnew[i] = PRF_GRSUM1(pwrnew[i]);

        /* Determine which columns to show */
        int i0 = 100;
        int iN = 0;
        for (i = 0;i < col;i++)
        {
                int i0_temp, iN_temp;

                /* Column has mixed phase */
                if (0 < eofsum[i] / count_total[i] && eofsum[i] / count_total[i] < 1)
                {
                        i0_temp = i - 2;
                        if (i0_temp < i0) i0 = i0_temp;
                        iN_temp = i + 2;
                        if (iN_temp > iN) iN = iN_temp;
                }

                /* Column is between cells */
                if (i < col)
                        if (eofhigh[i + 1] == 0 && eoflow[i] == 1)
                        {
                                i0_temp = i - 2;
                                if (i0_temp < i0) i0 = i0_temp;
                                iN_temp = i + 1;
                                if (iN_temp > iN) iN = iN_temp;
                        }

                if (i0 < 0) i0 = 0;
                if (iN > col - 1) iN = col - 1;
        }

        /* Report Values */
        Message0("\nInterface Report:\n");
        Message0("   Column: ");
        for (i = i0;i <= iN; i++) Message0("       %2d ", i);
        Message0("\n");
        Message0("  Cell Ct: ");
        for (i = i0;i <= iN; i++) Message0("     %4d ", count_total[i]);
        Message0("\n");
        Message0(" Centroid: ");
        for (i = i0;i <= iN; i++) Message0("  %#7.5g ", centroid[i] / count_total[i]);
        Message0("\n");
        Message0("    Power: ");
        for (i = i0;i <= iN; i++) Message0(" %#+8.6g ", pwrnew[i] / count_total[i]);
        Message0("\n");
        Message0("  Pwr Chg: ");
        for (i = i0;i <= iN; i++) Message0(" %#+.5f ", (pwrnew[i] - pwrold[i]) / count_total[i]);
        Message0("\n");
        Message0("  EOF Avg: ");
        for (i = i0;i <= iN; i++) Message0("  %#.5f ", eofsum[i] / count_total[i]);
        Message0("\n");
        Message0("  EOF Min: ");
        for (i = i0;i <= iN; i++) Message0("  %#.5f ", eoflow[i]);
        Message0("\n");
        Message0("  EOF Max: ");
        for (i = i0;i <= iN; i++) Message0("  %#.5f ", eofhigh[i]);
        Message0("\n");
        Message0("    df/dt: ");
        for (i = i0;i <= iN; i++) Message0(" %#+8.1e ", dfdtsum[i] / count_total[i]);
        Message0("\n");

        free(dfdtsum);
        free(eofsum);
        free(eofhigh);
        free(eoflow);
        free(count_total);
        free(pwrold);
```

```
            free(pwrnew);
            free(centroid);
        }

        Message0("\n");

        /* Store for next time step*/
        begin_c_loop(c, t)
        {
            eof_m1(c, t) = eof(c, t);
            e_m1(c, t) = e(c, t);
            e_sen_m1(c, t) = e_sen(c, t);
            cv_m1(c, t) = cv(c, t);
            rho_m1(c, t) = rho(c, t);
        }
        end_c_loop(c, t)
        PRF_GSYNC();
#endif
}

DEFINE_REPORT_DEFINITION_FN(timestep_ender)
{
        node_to_host_int_1(iter);
        return iter;
}
DEFINE_REPORT_DEFINITION_FN(report_mass_liq)
{
        node_to_host_real_1(mass_liq);
        return mass_liq;
}
DEFINE_REPORT_DEFINITION_FN(report_mass_vap)
{
        node_to_host_real_1(mass_vap);
        return mass_vap;
}
DEFINE_REPORT_DEFINITION_FN(report_real_mass_tot)
{
        node_to_host_real_1(real_mass_tot);
        return real_mass_tot;
}
DEFINE_REPORT_DEFINITION_FN(report_real_mass_liq)
{
        node_to_host_real_1(real_mass_liq);
        return real_mass_liq;
}
DEFINE_REPORT_DEFINITION_FN(report_real_mass_vap)
{
        node_to_host_real_1(real_mass_vap);
        return real_mass_vap;
}
DEFINE_REPORT_DEFINITION_FN(report_TVbulk)
{
        node_to_host_real_1(T_vap_bulk);
        return T_vap_bulk;
}
DEFINE_REPORT_DEFINITION_FN(report_rhoVbulk)
{
        node_to_host_real_1(rho_vap_bulk);
        return rho_vap_bulk;
}
DEFINE_REPORT_DEFINITION_FN(report_ELF)
{
        node_to_host_real_1(elf);
        return elf;
}
DEFINE_REPORT_DEFINITION_FN(report_Psat)
{
        node_to_host_real_1(P_sat);
        return P_sat;
}
DEFINE_REPORT_DEFINITION_FN(report_Tsat)
{
        node_to_host_real_1(T_sat);
        return T_sat;
}
DEFINE_REPORT_DEFINITION_FN(report_gravity)
{
        node_to_host_real_1(gravity);
        return gravity;
}
DEFINE_REPORT_DEFINITION_FN(report_rho0)
{
        node_to_host_real_1(rho0);
        return rho0;
}
DEFINE_REPORT_DEFINITION_FN(report_interf)
{
        node_to_host_real_1(interf_new);
        return interf_new;
}
DEFINE_REPORT_DEFINITION_FN(report_hf_dry_wall)
{
        node_to_host_real_1(HF_dry);
        return HF_dry;
}
DEFINE_REPORT_DEFINITION_FN(report_hf_wet_wall)
{
        node_to_host_real_1(HF_wet);
        return HF_wet;
}
DEFINE_REPORT_DEFINITION_FN(report_hf_fwd_dome)
{
        node_to_host_real_1(HF_fwd);
        return HF_fwd;
}
DEFINE_REPORT_DEFINITION_FN(report_hf_aft_dome)
{
        node_to_host_real_1(HF_aft);
        return HF_aft;
}
DEFINE_REPORT_DEFINITION_FN(report_hr_dry_wall)
{
        node_to_host_real_1(HR_dry);
        return HR_dry;
}
DEFINE_REPORT_DEFINITION_FN(report_hr_wet_wall)
```

```
{
        node_to_host_real_1(HR_wet);
        return HR_wet;
}
DEFINE_REPORT_DEFINITION_FN(report_hr_fwd_dome)
{
        node_to_host_real_1(HR_fwd);
        return HR_fwd;
}
DEFINE_REPORT_DEFINITION_FN(report_hr_aft_dome)
{
        node_to_host_real_1(HR_aft);
        return HR_aft;
}
DEFINE_REPORT_DEFINITION_FN(report_hi_dry_wall)
{
        node_to_host_real_1(HI_dry);
        return HI_dry;
}
DEFINE_REPORT_DEFINITION_FN(report_hi_wet_wall)
{
        node_to_host_real_1(HI_wet);
        return HI_wet;
}
DEFINE_REPORT_DEFINITION_FN(report_hi_fwd_dome)
{
        node_to_host_real_1(HI_fwd);
        return HI_fwd;
}
DEFINE_REPORT_DEFINITION_FN(report_hi_aft_dome)
{
        node_to_host_real_1(HI_aft);
        return HI_aft;
}
```

APPENDIX I CONTINUED FLUENT SETUP FOR NORMAL GRAVITY CASE

This appendix details the case-specific setup of the case found in the Normal Gravity Small Scale Terrestrial Spherical Tank section. It is a continuation of the general instructions presented in Appendix F. The source code for this case is provided in Appendix J.

1. Starting from the last item of Appendix F, assign the boundary conditions.
    a. In the *Tree, Setup → Boundary Conditions*.
    b. For *wall-dry*, open the respective *Wall* dialog box and go to the *UDS* tab.
    c. Ensure the *User-Defined Scalar Boundary Condition* is set to *Specified Flux*.
    d. Set the *User-Defined Scalar Boundary Value* to the dry wall flux in the dropdown menu.
    e. The same procedure is done to *wall-wet* using the wet wall flux.
2. Set up the temperature rakes.
    a. In the ribbon, *Setting Up Domain → Surface → Create → Line/Rake…* to open the *Line/Rake Surface* dialog box.
    b. The axial locations of the temperature transducers are given in Table 18.

Table 18 Axial locations of temperature rakes.

Temperature Transducer Designation	Distance from Top of Tank, m
R-2	0.08191
R-3	0.06702
R-4	0.04989
R-5	0.03537

3. Create the report definitions for the temperature rakes.
    a. In the *Tree, Solution → Report Definitions → New → Surface Report → Area-weighted Average…* to open the *Surface Report Definition* dialog box.
    b. Name the report definition accordingly and set the *Field Variable* to *User Defined Scalars → Scalar-0*.
    c. Select the desired temperature rake. For a more concise format, select the *Per Surface* option under *Options* and select all desired temperature rakes. This will make a single report definition that will output the individual area-weighted average temperatures of the rake surfaces.
4. Initialize the solution.
    a. In the *Tree, Solution → Initialization → Initialize*.
    b. Ensure the starting fill level is close to 50.5%.
5. Set the transient calculation options.
    a. In the *Tree, Solution → Run Calculation* to activate the *Run Calculation Task Page*.
    b. Choose a *Time Step Size* of 0.01 sec.
    c. Set the *Number of Time Steps* to 100000.
    d. Set *Max Iterations/Time Step* to at least 10000.
    e. Set a *Reporting Interval* to 10 or leave as 1.
6. Save the case again and set up autosaves and animations if desired.

7. Run the simulation.
    a. In the *Tree*, *Solution* → *Run Calculation*.
    b. Press *Calculate* to run.

APPENDIX J SOURCE CODE FOR NORMAL GRAVITY CASE

```
#include "udf.h"
#include "sg.h" /* This is needed for function BOUNDARY_FACE_GEOMETRY */
#include <stdio.h>
#include <stdlib.h>

/* External Function Prototypes */
extern double meanof(double, double, double);
extern double prop(double, double, double, int);
extern double TsatfPsat(double, double);
extern void PsatfTsat(double, double*, double*);
extern double PfTR(double, double);
extern void RfTsat(double, double*, double*);

/* Pointer Variables */
double pk1, pk2, prho1, prho2, pe1, pe2, pel, pev, pmu1, pmu2, pcv1, pcv2;

/* Thread Indices */
#define fluid 5
#define i_dry 7
#define i_adi 8
#define i_wet 9
#define i_start 7
#define i_end 9

/* Wall Variables */
double A_wall[10];
#define A_dry A_wall[i_dry]    /* Ullage (dry) Wall */
#define A_adi A_wall[i_adi]    /* Adiabatic Wall */
#define A_wet A_wall[i_wet]    /* Liquid (wet) Wall */

/* Heat flux */
double HF_wall[10];
#define HF_dry HF_wall[i_dry]
#define HF_adi HF_wall[i_adi]
#define HF_wet HF_wall[i_wet]

/* Heat rate */
double HR_wall[10];
#define HR_dry HR_wall[i_dry]
#define HR_adi HR_wall[i_adi]
#define HR_wet HR_wall[i_wet]

/* Heat input */
double HI_wall[10];
#define HI_dry HI_wall[i_dry]
#define HI_adi HI_wall[i_adi]
#define HI_wet HI_wall[i_wet]

/* Constants */
double interf = 0.114200;/* axial location [m] of liq-vap interface; fill level should be 50.5% */
double P_init = 101325;  /* Initial Pressure */
double T_init = 20.27;     /* Initial Guessed Temperature */
double geom_mult = 2 * M_PI; /* Volume and area values are multiplied by this to find true volume/area */

/* User Defined Scalar */
#define T(c, t) C_UDSI(c, t, 0)
#define T_m1(c, t) C_UDSI_M1(c, t, 0)
#define FT(f, t) F_UDSI(f, t, 0)

/* User Defined Memory Variables */
#define eof(c, t) C_UDMI(c, t, 0)       /* Phase Fraction */
#define e(c, t) C_UDMI(c, t, 1)         /* Internal Energy [J/kg] */
#define e_sen(c, t) C_UDMI(c, t, 2)     /* Sensible Heat [J/kg] */
#define cv(c, t) C_UDMI(c, t, 3)        /* Specific Heat [J/kg-K] */
#define rho(c, t) C_UDMI(c, t, 4)       /* Density [kg/m3] */
#define k(c, t) C_UDMI(c, t, 5)         /* Thermal Conductivity [W/m-K] */
#define mu(c, t) C_UDMI(c, t, 6)        /* Viscosity [kg/m-s] */

/* Fluid properties that need to be stored for autosaves */
#define C_mass_tot(c, t) C_UDMI(c, t, 7)
#define C_mass_vap_t0(c, t) C_UDMI(c, t, 8)
#define C_mass_liq_t0(c, t) C_UDMI(c, t, 9)
#define C_P_sat(c, t) C_UDMI(c, t, 10)
#define C_T_sat(c, t) C_UDMI(c, t, 11)
#define C_delta_e(c, t) C_UDMI(c, t, 12)

/* Fluid properties from last time step */
#define eofold(c, t) C_UDMI(c, t, 13)
#define eof_m1(c, t) C_UDMI(c, t, 14)
#define e_m1(c, t) C_UDMI(c, t, 15)
#define e_sen_m1(c, t) C_UDMI(c, t, 16)
#define cv_m1(c, t) C_UDMI(c, t, 17)
#define rho_m1(c, t) C_UDMI(c, t, 18)

/* Extra UDMs */
#define C_interface(c, t) C_UDMI(c, t, 19)
#define C_search(c, t) C_UDMI(c, t, 20)
#define C_power(c, t) C_UDMI(c, t, 21)

/* For Axisymmetric Geometries */
#define vol(c, t) geom_mult*C_VOLUME(c, t)

/* Residuals */
double tol_mass = 1e-4, tol_vel = 1e-6, tol_uds0 = 1e-6, tol_eof = 1e-6, tol_p = 1e-6, tol_elf = 1e-6;
double res[7];
#define r_mass res[0]
#define r_xvel res[1]
#define r_yvel res[2]
#define r_temp res[3]
#define r_eof res[4]
#define r_pres res[5]
#define r_elf res[6]
double r_eof1;
double r_eof2;
char res_label[7][11] = {
        "continuity",
        "x-velocity",
        "y-velocity",
        "    energy",
        "       eof",
```

```
        "  pressure",
        "       elf"
};

/* Fluid properties shown in residuals */
double sat[8];
#define T_vap_bulk sat[0]       /* Bulk Vapor Temperature */
#define rho_vap_bulk sat[1]     /* Bulk Vapor Density */
#define P_sat sat[2]            /* Saturation Pressure */
#define T_sat sat[3]            /* Saturation Temperature */
#define elf sat[4]              /* Evaporated Liquid Fraction */
#define delta_e sat[5]          /* Latent Heat */
#define T_liq_bulk sat[6]/* Bulk Liquid Temperature */
#define rho_liq_bulk sat[7]     /* Bulk Liquid Density */
char sat_label[8][9] = {
        "T-V-Bulk",
        "R-V-Bulk",
        "Sat.Pres",
        "Sat.Temp",
        "EvapLiqF",
        "Lat.Heat",
        "T-L-Bulk",
        "R-L-Bulk"
};

/* Fluid Properties not shown in residuals */
double mass_tot;            /* Total Mass at Initialization (Stays Constant) */
double mass_liq;            /* Mass of Liquid Phase at Current Time */
double mass_liq_t0;             /* Mass of Liquid Phase at Initialization */
double mass_vap;            /* Mass of Vapor Phase at Current Time */
double mass_vap_t0;             /* Mass of Vapor Phase at Initialization */
double vol_tot;                 /* Total domain volume */
double fill;                /* Percent fill of tank */
double T_liq_sum, vol_liq_sum;      /* Bulk liquid properties */
double T_vap_sum, vol_vap_sum;      /* Bulk vapor properties */
double real_mass_tot, real_mass_vap, real_mass_liq;     /* Masses based on density (Vapor mass calculated
from vapor density) */
double max_liq_T, min_vap_T;/* Maximum temperatures found in fluid domain */
double rho0;

/* Iteration Variables and Parameters */
double resid;       /* temporary residual */
double sum;         /* temporary summation */
double eofsum;                  /* summation of mixed eofs for residual calculation */
double eofchi = 0.7;        /* eof relaxation factor */
double P_sat_chi = 0.3;   /* pressure relaxation factor */
int flag;       /* switch to trigger timestep convergence report definition */
int iter, res_count, count; /* iteration counters */
double dT = 0.1;    /* temperature interval used for energy derivative */
double interf_new, cell_size, x_start;
int col;

DEFINE_EXECUTE_ON_LOADING(execute_on_loading, libname)
{
        iter = 0;
#if !RP_HOST
        Domain *d = Get_Domain(1);
        Thread *t = Lookup_Thread(d, fluid);
        Thread *t_dry = Lookup_Thread(d, i_dry);
        Thread *t_adi = Lookup_Thread(d, i_adi);
        Thread *t_wet = Lookup_Thread(d, i_wet);
        Thread *tt;
        cell_t c;
        face_t f;
        int i;

        /* Calculate Wall Areas */
        Message0("\nCalculating Wall Areas...\n");
        real Area[ND_ND], ds, es[ND_ND], A_by_es, dr0[ND_ND];
        for (i = i_start;i <= i_end;i++) A_wall[i] = 0;
        begin_c_loop_int(c, t)
        {
                c_face_loop(c, t, i)
                {
                        f = C_FACE(c, t, i);
                        tt = C_FACE_THREAD(c, t, i);
                        if (BOUNDARY_FACE_THREAD_P(tt))
                        {
                                BOUNDARY_FACE_GEOMETRY(f, tt, Area, ds, es, A_by_es, dr0);
                                if (tt == t_dry) A_dry += NV_MAG(Area);
                                if (tt == t_wet) A_wet += NV_MAG(Area);
                                if (tt == t_adi) A_adi += NV_MAG(Area);
                        }
                }
        }
        end_c_loop_int(c, t)

        /* Sum all values across compute nodes */
        for (i = i_start;i <= i_end;i++) A_wall[i] = geom_mult * PRF_GRSUM1(A_wall[i]);

        Message0("\nWall Areas:\n");
        Message0("Ullage Wall     %8.7g m2\n", A_dry);
        Message0("Adiabatic Wall  %8.7g m2\n", A_adi);
        Message0("Liquid Wall     %8.7g m2\n", A_wet);
        Message0("Total Area for Walls Receiving Heat %8.7g m2\n", (A_dry + A_wet));
        if (A_dry == 0 || A_wet == 0) Message0("\nERROR!\nThe wall areas must not be 0.\nActivate
'calculate_wall_areas' in Execute on Demand before proceeding.\n");
#endif
        node_to_host_real_3(A_dry, A_wet, A_adi);
}

DEFINE_ON_DEMAND(calculate_wall_areas)
{
#if !RP_HOST
        Domain *d = Get_Domain(1);
        Thread *t = Lookup_Thread(d, fluid);
        Thread *t_dry = Lookup_Thread(d, i_dry);
        Thread *t_adi = Lookup_Thread(d, i_adi);
        Thread *t_wet = Lookup_Thread(d, i_wet);
        Thread *tt;
        cell_t c;
        face_t f;
        int i;

        /* Calculate Wall Areas */
        Message0("\nCalculating Wall Areas...\n");
```

```
        real Area[ND_ND], ds, es[ND_ND], A_by_es, dr0[ND_ND];
        for (i = i_start;i <= i_end;i++) A_wall[i] = 0;
        begin_c_loop_int(c, t)
        {
                c_face_loop(c, t, i)
                {
                        f = C_FACE(c, t, i);
                        tt = C_FACE_THREAD(c, t, i);
                        if (BOUNDARY_FACE_THREAD_P(tt))
                        {
                                BOUNDARY_FACE_GEOMETRY(f, tt, Area, ds, es, A_by_es, dr0);
                                if (tt == t_dry) A_dry += NV_MAG(Area);
                                if (tt == t_adi) A_adi += NV_MAG(Area);
                                if (tt == t_wet) A_wet += NV_MAG(Area);
                        }
                }
        }
        end_c_loop_int(c, t)

        /* Sum all values across compute nodes */
        for (i = i_start;i <= i_end;i++) A_wall[i] = geom_mult * PRF_GRSUM1(A_wall[i]);

        Message0("\nWall Areas:\n");
        Message0("Ullage Wall     %8.7g m2\n", A_dry);
        Message0("Adiabatic Wall  %8.7g m2\n", A_adi);
        Message0("Liquid Wall     %8.7g m2\n", A_wet);
        Message0("Total Area for Walls Receiving Heat %8.7g m2\n", (A_dry + A_wet));
        if (A_dry == 0 || A_wet == 0) Message0("The wall areas must not be 0.\nActivate
'calculate_wall_areas' in Execute on Demand before proceeding.\n");
#endif
        node_to_host_real_3(A_dry, A_wet, A_adi);
}

DEFINE_INIT(hook_init, d)
{
#if !RP_HOST
        Message0("\nBeginning Initialization...\n");
        if (A_dry == 0 || A_wet == 0) Message0("Activate 'calculate_wall_areas' in Execute on Demand
before proceeding.\n");

        Thread *t = Lookup_Thread(d, fluid);
        Thread *t_dry = Lookup_Thread(d, i_dry);
        Thread *t_adi = Lookup_Thread(d, i_adi);
        Thread *t_wet = Lookup_Thread(d, i_wet);
        cell_t c;
        face_t f;
        Node *node;
        int i;

        real x[ND_ND];

        /* Reset All Values */
        iter = 0;
        eofsum = 1;

        interf_new = interf;

        /* Patch EOF */
        begin_c_loop(c, t)
        {
                C_CENTROID(x, c, t);
                real max = -1e6;
                real min = 1e6;
                c_node_loop(c, t, i)
                {
                        node = C_NODE(c, t, i);
                        if (NODE_X(node) > max) max = NODE_X(node);
                        if (NODE_X(node) < min) min = NODE_X(node);
                }
                if (max > interf && min > interf) eof(c, t) = 0;
                if (max > interf && min < interf) eof(c, t) = (interf - min) / (max - min);
                if (max < interf && min < interf) eof(c, t) = 1;
        }
        end_c_loop(c, t)

        /* Mark search zone */
        begin_c_loop(c, t)
        {
                C_search(c, t) = 0;
                C_CENTROID(x, c, t);
                if ((interf-0.0085 < x[0]) && (x[0] < interf+0.0035)) C_search(c, t) = 1;
        }
        end_c_loop(c, t)

        /* Mark interface cells */
        begin_c_loop(c, t)
        {
                if ((0 < eof(c, t)) && (eof(c, t) < 1) && (C_search(c, t)))
                {
                        C_interface(c, t) = 1;
                        C_power(c, t) = 16;
                }
                else
                {
                        C_interface(c, t) = 0;
                        C_power(c, t) = -4;
                }
        }
        end_c_loop(c, t)

        /* Calculate cell size */
        sum = 0;
        count = 0;
        Message0("\nCalculating Cell Sizes...\n");
        begin_c_loop_int(c, t)
        {
                if (C_search(c, t))
                {
                        C_CENTROID(x, c, t);
                        sum += vol(c, t) / (2 * M_PI*x[1]);
                        count++;
                }
        }
        end_c_loop_int(c, t)
        sum = PRF_GRSUM1(sum);
        count = PRF_GISUM1(count);
```

```
cell_size = sqrt(sum / count);
Message0(" Average size of %d cells: %10.4e m\n", count, cell_size);
cell_size = 1e-4*round(cell_size / 1e-4);
Message0(" Rounded Cell Size: %10.4e m\n", cell_size);

col = round(0.012 / cell_size);
Message0(" %d Columns in search zone\n", col);
x_start = interf - 0.0085;

/* Calculate Saturation Properties */
P_sat = P_init;
T_init = TsatfPsat(P_init, T_init);
T_sat = T_init;
RfTsat(T_sat, &prho1, &prho2);
pel = prop(T_sat, prho1, P_sat, 2);
pev = prop(T_sat, prho2, P_sat, 2);
delta_e = pev - pel;

/* Patch Temperatures */
begin_c_loop(c, t)
{
	T(c, t) = T_sat;
}
end_c_loop(c, t)
begin_f_loop(f, t_dry)
{
	FT(f, t_dry) = T_sat;
}
end_f_loop(f, t_dry)
begin_f_loop(f, t_wet)
{
	FT(f, t_wet) = T_sat;
}
end_f_loop(f, t_wet)
begin_f_loop(f, t_adi)
{
	FT(f, t_adi) = T_sat;
}
end_f_loop(f, t_adi)

/* Initialize Flow Properties */
begin_c_loop(c, t)
{
	/* Liquid */
	if (eof(c, t) <= 0)
	{
		rho(c, t) = prop(T(c, t), 70.0, P_sat, 1);
		e(c, t) = prop(T(c, t), rho(c, t), P_sat, 2);
		cv(c, t) = prop(T(c, t), rho(c, t), P_sat, 3);
		k(c, t) = prop(T(c, t), rho(c, t), P_sat, 4);
		mu(c, t) = prop(T(c, t), rho(c, t), P_sat, 5);
	}
	/* Mixed Phase */
	if ((eof(c, t) > 0) && (eof(c, t) < 1))
	{
		RfTsat(T(c, t), &prho1, &prho2);
		rho(c, t) = meanof(prho1, prho2, eof(c, t));
		pe1 = prop(T(c, t), prho1, P_sat, 2);
		pe2 = prop(T(c, t), prho2, P_sat, 2);
		e(c, t) = meanof(pe1, pe2, eof(c, t));

		pcv1 = prop(T(c, t), prho1, P_sat, 3);
		pcv2 = prop(T(c, t), prho2, P_sat, 3);
		cv(c, t) = meanof(pcv1, pcv2, eof(c, t));

		pk1 = prop(T(c, t), prho1, P_sat, 4);
		pk2 = prop(T(c, t), prho2, P_sat, 4);
		k(c, t) = meanof(pk1, pk2, eof(c, t));

		pmu1 = prop(T(c, t), prho1, P_sat, 5);
		pmu2 = prop(T(c, t), prho2, P_sat, 5);
		mu(c, t) = meanof(pmu1, pmu2, eof(c, t));
	}
	/* Vapor */
	if (eof(c, t) >= 1)
	{
		rho(c, t) = prop(T(c, t), 1.0, P_sat, 1);
		e(c, t) = prop(T(c, t), rho(c, t), P_sat, 2);
		cv(c, t) = prop(T(c, t), rho(c, t), P_sat, 3);
		k(c, t) = prop(T(c, t), rho(c, t), P_sat, 4);
		mu(c, t) = prop(T(c, t), rho(c, t), P_sat, 5);
	}
	/* Sensible Heat */
	e_sen(c, t) = e(c, t) - eof(c, t)*delta_e;

	/* Previous timestep values */
	eofold(c, t) = eof(c, t);
	eof_m1(c, t) = eof(c, t);
	e_m1(c, t) = e(c, t);
	e_sen_m1(c, t) = e_sen(c, t);
	cv_m1(c, t) = cv(c, t);
	rho_m1(c, t) = rho(c, t);
}
end_c_loop(c, t)

/* Calculate System Mass and Bulk Properties */
mass_tot = 0, vol_tot = 0, mass_liq_t0 = 0, T_liq_sum = 0, vol_liq_sum = 0, T_vap_sum = 0,
vol_vap_sum = 0;
begin_c_loop_int(c, t) /* Interior cell loop prevents double counting of cells between domain
partitions */
{
	mass_tot += rho(c, t) * vol(c, t);
	vol_tot += vol(c, t);
	if (eof(c, t) != 1)
	{
		mass_liq_t0 += prop(T(c, t), 70.0, P_sat, 1) * vol(c, t)*(1 - eof(c, t));
		T_liq_sum += T(c, t)*vol(c, t)*(1 - eof(c, t));
		vol_liq_sum += vol(c, t)*(1 - eof(c, t));
	}
	if (eof(c, t) != 0)
	{
		T_vap_sum += T(c, t)*vol(c, t)*eof(c, t);
		vol_vap_sum += vol(c, t)*eof(c, t);
	}
}
```

```
        end_c_loop_int(c, t)
        /* PRF_GRSUM performs summations across all processing nodes */
        mass_tot = PRF_GRSUM1(mass_tot);
        mass_liq_t0 = PRF_GRSUM1(mass_liq_t0);
        mass_vap_t0 = mass_tot - mass_liq_t0;
        vol_tot = PRF_GRSUM1(vol_tot);
        /* Calculate bulk properties */
        mass_liq = mass_liq_t0;
        T_liq_sum = PRF_GRSUM1(T_liq_sum);
        vol_liq_sum = PRF_GRSUM1(vol_liq_sum);
        T_liq_bulk = T_liq_sum / vol_liq_sum;
        rho_liq_bulk = mass_liq_t0 / vol_liq_sum;
        mass_vap = mass_vap_t0;
        T_vap_sum = PRF_GRSUM1(T_vap_sum);
        vol_vap_sum = PRF_GRSUM1(vol_vap_sum);
        T_vap_bulk = T_vap_sum / vol_vap_sum;
        rho_vap_bulk = mass_vap_t0 / vol_vap_sum;
        fill = 100 * vol_liq_sum / vol_tot;

        /* Export to UDM */
        begin_c_loop(c, t)
        {
                C_P_sat(c, t) = P_sat;
                C_T_sat(c, t) = T_sat;
                C_delta_e(c, t) = delta_e;
                C_mass_tot(c, t) = mass_tot;
                C_mass_vap_t0(c, t) = mass_vap_t0;
                C_mass_liq_t0(c, t) = mass_liq_t0;
        }
        end_c_loop(c, t)

        /* Calculate real masses */
        real_mass_tot = 0, real_mass_vap = 0, real_mass_liq = 0;
        begin_c_loop_int(c, t)
        {
                real_mass_tot += rho(c, t)*vol(c, t);
                if (eof(c, t) != 0) real_mass_vap += prop(T(c, t), 1.0, P_sat, 1)*vol(c, t)*eof(c, t);
                if (eof(c, t) != 1) real_mass_liq += prop(T(c, t), 70.0, P_sat, 1)*vol(c, t)*(1 - eof(c, t));
        }
        end_c_loop_int(c, t)
        real_mass_tot = PRF_GRSUM1(real_mass_tot);
        real_mass_vap = PRF_GRSUM1(real_mass_vap);
        real_mass_liq = PRF_GRSUM1(real_mass_liq);
        rho0 = real_mass_vap / vol_vap_sum;

        /* Display Initialized Data */
        Message0("\nInitialized Flow Variables:\n\n");
        Message0(" Total Mass  Liquid Mass  Vapor Mass\n");
        Message0(" %#10.4e   %#10.4e  %#10.4e\n\n", mass_tot, mass_liq_t0, mass_vap_t0);
        Message0(" Vapor Vol.  T-V-Bulk  R-V-Bulk ");
        Message0(" Liquid Vol  T-L-Bulk  R-L-Bulk\n");
        Message0(" %#10.4e  %#8.7g  %#8.7g ", vol_vap_sum, T_vap_bulk, rho_vap_bulk);
        Message0(" %#10.4e  %#8.7g  %#8.7g\n\n", vol_liq_sum, T_liq_bulk, rho_liq_bulk);
        Message0(" Sat.Pres  Sat.Temp  Lat.Heat\n");
        Message0(" %#8.7g  %#8.7g  %#8.7g\n", P_sat, T_sat, delta_e);
        Message0("\nPercent Fill: %#4.3g%%\n", fill);
        Message0("\nInitialization Complete!\n");
#endif
}

/**************************
Start FLUENT Iteration Loop
**************************/

DEFINE_PROFILE(profile_flux_dry_wall, t, i)
{
        face_t f;
        HF_dry = 101.0;
        begin_f_loop(f, t)
        {
                F_PROFILE(f, t, i) = HF_dry;
        }
        end_f_loop(f, t)
}
DEFINE_PROFILE(profile_flux_wet_wall, t, i)
{
        face_t f;
        HF_wet = 7.0;
        begin_f_loop(f, t)
        {
                F_PROFILE(f, t, i) = HF_wet;
        }
        end_f_loop(f, t)
}

DEFINE_ADJUST(hook_adjust, d)
{
#if !RP_HOST
        Thread *t = Lookup_Thread(d, fluid);
        cell_t c;
        int i;

        if (iter == 0)
        {
                r_eof = 0;
                r_eof1 = 0;
                r_eof2 = 0;
                r_pres = 0;
                r_elf = 0;
                res_count = 0;
        }

        int iter_old = iter;
        iter++;
        flag = 0;
        /* Obtain residual */
        if (iter > 1)
        {
                for (i = 0; i <= 3; i++)
                {
                        if (DOMAIN_RES_SCALE(d, i) == 0) res[i] = DOMAIN_RES(d, i);
                        else res[i] = DOMAIN_RES(d, i) / DOMAIN_RES_SCALE(d, i);
                }
        }

        /* Import from UDM */
```

```
begin_c_loop(c, t)
{
	mass_tot = C_mass_tot(c, t);
	mass_vap_t0 = C_mass_vap_t0(c, t);
	mass_liq_t0 = C_mass_liq_t0(c, t);
	P_sat = C_P_sat(c, t);
	T_sat = C_T_sat(c, t);
	delta_e = C_delta_e(c, t);
}
end_c_loop(c, t)

PRF_GSYNC();
/* Calculate Mass and Bulk Vapor Properties */
mass_liq = 0, T_liq_sum = 0, vol_liq_sum = 0, mass_vap = 0, T_vap_sum = 0, vol_vap_sum = 0;
begin_c_loop_int(c, t)
{
	if (eof(c, t) != 1)
	{
		mass_liq += prop(T(c, t), 70.0, P_sat, 1)*vol(c, t)*(1 - eof(c, t));
		T_liq_sum += T(c, t)*vol(c, t)*(1 - eof(c, t));
		vol_liq_sum += vol(c, t)*(1 - eof(c, t));
	}
	if (eof(c, t) != 0);
	{
		T_vap_sum += T(c, t)*vol(c, t)*eof(c, t);
		vol_vap_sum += vol(c, t)*eof(c, t);
	}
}
end_c_loop_int(c, t)
mass_liq = PRF_GRSUM1(mass_liq);
T_liq_sum = PRF_GRSUM1(T_liq_sum);
vol_liq_sum = PRF_GRSUM1(vol_liq_sum);
T_liq_bulk = T_liq_sum / vol_liq_sum;
rho_liq_bulk = mass_liq / vol_liq_sum;
mass_vap = mass_tot - mass_liq;
T_vap_sum = PRF_GRSUM1(T_vap_sum);
vol_vap_sum = PRF_GRSUM1(vol_vap_sum);
T_vap_bulk = T_vap_sum / vol_vap_sum;
rho_vap_bulk = mass_vap / vol_vap_sum;
fill = 100 * vol_liq_sum / vol_tot;

/* Obtain real masses */
PRF_GSYNC();
real_mass_tot = 0, real_mass_vap = 0, real_mass_liq = 0;
begin_c_loop_int(c, t)
{
	real_mass_tot += rho(c, t)*vol(c, t);
	if (eof(c, t) != 0) real_mass_vap += prop(T(c, t), 1.0, P_sat, 1)*vol(c, t)*eof(c, t);
	if (eof(c, t) != 1) real_mass_liq += prop(T(c, t), 70.0, P_sat, 1)*vol(c, t)*(1 - eof(c, t));
}
end_c_loop_int(c, t)
real_mass_tot = PRF_GRSUM1(real_mass_tot);
real_mass_vap = PRF_GRSUM1(real_mass_vap);
real_mass_liq = PRF_GRSUM1(real_mass_liq);
rho0 = real_mass_vap / vol_vap_sum;

/* Calculate saturation pressure */
real P_sat_old = P_sat; /* Store pressure for residual */
P_sat = PfTR(T_vap_bulk, rho_vap_bulk); /* Update pressure using bulk vapor properties */
r_pres = fabs(P_sat - P_sat_old) / P_sat_old; /* Calculate pressure residual */
P_sat = (P_sat_chi)*P_sat + (1 - P_sat_chi)*P_sat_old; /* Relax pressure */

/* Calculate saturation temperature */
T_sat = TsatfPsat(P_sat, T_sat);

/* Calculate saturation energies and latent heat */
RfTsat(T_sat, &prho1, &prho2);
pel = prop(T_sat, prho1, P_sat, 2);
pev = prop(T_sat, prho2, P_sat, 2);
delta_e = pev - pel;

/* Calculate evaporated liquid fraction */
real elf_old = elf;
elf = 1 - (mass_liq / mass_liq_t0);
r_elf = fabs((elf - elf_old) / elf_old);

/* Calculate Density, Internal Energy, and EOF */
PRF_GSYNC();
begin_c_loop(c, t)
{
	/* Determine density and energy using new temperature and old eof */
	if (eof(c, t) == 0)
	{
		rho(c, t) = prop(T(c, t), rho(c, t), P_sat, 1);
		e(c, t) = prop(T(c, t), rho(c, t), P_sat, 2);
	}
	if (eof(c, t) > 0 && eof(c, t) < 1)
	{
		RfTsat(T(c, t), &prho1, &prho2);
		rho(c, t) = meanof(prho1, prho2, eof(c, t));

		pe1 = prop(T(c, t), prho1, P_sat, 2);
		pe2 = prop(T(c, t), prho2, P_sat, 2);
		e(c, t) = meanof(pe1, pe2, eof(c, t));
	}
	if (eof(c, t) == 1)
	{
		rho(c, t) = prop(T(c, t), rho(c, t), P_sat, 1);
		e(c, t) = prop(T(c, t), rho(c, t), P_sat, 2);
	}
	e_sen(c, t) = e(c, t) - eof(c, t)*delta_e;

	/* Save old eof to calculate residual */
	eofold(c, t) = eof(c, t);

	/* Determine new eof using energy */
	if (e(c, t) > pel && e(c, t) < pev) eof(c, t) = (e(c, t) - pel) / delta_e;
	if (e(c, t) <= pel) eof(c, t) = 0;
	if (e(c, t) >= pev) eof(c, t) = 1;
}
end_c_loop(c, t)

/* Calculate eof residual */
PRF_GSYNC();
real eofsumold = eofsum;
resid = 0, r_eof1 = 0, r_eof2 = 0, eofsum = 1;
```

```
begin_c_loop_int(c, t)
{
	/* Method 1: Maximum Cell Change */
	if (eofold(c, t) <= 0) resid = eof(c, t);
	else resid = fabs(eofold(c, t) - eof(c, t) / eofold(c, t));
	if (resid > r_eof1) r_eof1 = resid;

	/* Method 2: Change in the Sum of Mushy Zone */
	if (eof(c, t) > 0 && eof(c, t) < 1) eofsum += eof(c, t);
}
end_c_loop_int(c, t)
r_eof1 = PRF_GRHIGH1(r_eof1);
eofsum = PRF_GRSUM1(eofsum);
r_eof2 = fabs(eofsum - eofsumold) / eofsumold;
r_eof = r_eof2; /* Choose method for residual */

/* Update properties from new eof and find max temps */
max_liq_T = 0.0, min_vap_T = 1000.0;
begin_c_loop(c, t)
{
	/* Relax EOF */
	eof(c, t) = eofchi * eof(c, t) + (1 - eofchi)*eofold(c, t);

	/* Liquid */
	if (eof(c, t) == 0)
	{
		cv(c, t) = prop(T(c, t), rho(c, t), P_sat, 3);
		k(c, t) = prop(T(c, t), rho(c, t), P_sat, 4);
		mu(c, t) = prop(T(c, t), rho(c, t), P_sat, 5);
	}
	/* Mixed Phase */
	if ((eof(c, t) > 0) && (eof(c, t) < 1))
	{
		RfTsat(T(c, t), &prho1, &prho2);

		pcv1 = prop(T(c, t), prho1, P_sat, 3);
		pcv2 = prop(T(c, t), prho2, P_sat, 3);
		cv(c, t) = meanof(pcv1, pcv2, eof(c, t));

		pk1 = prop(T(c, t), prho1, P_sat, 4);
		pk2 = prop(T(c, t), prho2, P_sat, 4);
		k(c, t) = meanof(pk1, pk2, eof(c, t));

		pmu1 = prop(T(c, t), prho1, P_sat, 5);
		pmu2 = prop(T(c, t), prho2, P_sat, 5);
		mu(c, t) = meanof(pmu1, pmu2, eof(c, t));
	}
	/* Vapor */
	if (eof(c, t) == 1)
	{
		cv(c, t) = prop(T(c, t), rho(c, t), P_sat, 3);
		k(c, t) = prop(T(c, t), rho(c, t), P_sat, 4);
		mu(c, t) = prop(T(c, t), rho(c, t), P_sat, 5);
	}

	/* Find min/max temperatures */
	if (eof(c, t) == 0 && T(c, t) > max_liq_T) max_liq_T = T(c, t);
	if (eof(c, t) == 1 && T(c, t) < min_vap_T) min_vap_T = T(c, t);
}
end_c_loop(c, t)
max_liq_T = PRF_GRHIGH1(max_liq_T);
min_vap_T = PRF_GRLOW1(min_vap_T);

/* Check convergence when pressure is below tolerance */
if (r_pres < tol_p) flag = 1;
if (r_elf > tol_elf) flag = 0;
if (r_eof > tol_eof) flag = 0;
if (r_temp > tol_uds0) flag = 0;
if (r_mass > tol_mass) flag = 0;
if (r_xvel > tol_vel) flag = 0;
if (r_yvel > tol_vel) flag = 0;
/* If the flag remains active after checking residuals,
then the report definition timestep_ender will trigger the end of the timestep for Fluent */

/* Export to UDM */
begin_c_loop(c, t)
{
	C_P_sat(c, t) = P_sat;
	C_T_sat(c, t) = T_sat;
	C_delta_e(c, t) = delta_e;
}
end_c_loop(c, t)

/* Display residual every reporting interval */
PRF_GSYNC();
if (iter%RP_Get_Integer("iteration-chunk") == 0)
{
	res_count++;
	if (res_count % 20 == 1)
	{
		Message0("\nTime = %g sec\n", CURRENT_TIME);
		Message0("  iter ");
		for (i = 0;i <= 6;i++) Message0(" %s ", res_label[i]);
		Message0("|");
		for (i = 0;i <= 3;i++) Message0(" %s ", sat_label[i]);
		Message0("  %s ", sat_label[4]); /* This one needs more space */
		Message0("  T-V-Min   T-L-Max ");
		Message0("\n");
	}
	Message0("%6d ", iter);
	for (i = 0;i <= 6;i++) Message0(" %10.4e ", res[i]);
	Message0("|");
	for (i = 0;i <= 3;i++) Message0(" %#8.7g ", sat[i]);
	Message0(" %+9.2e ", sat[4]);
	Message0(" %#8.7g  %#8.7g ", min_vap_T, max_liq_T);
	Message0("\n");
}

/* If flag says timestep has converged, then make iter value constant to end timestep */
if (flag == 1 && iter != 1 && iter%RP_Get_Integer("iteration-chunk") == 0)
{
	Message0("Solution converged in %d iterations\n", iter);
	for (i = 0;i <= 6;i++) Message0(" %s ", res_label[i]);
	Message0("\n");
	for (i = 0;i <= 6;i++) Message0(" %10.4e ", res[i]);
	Message0("\n");
```

```
            iter = iter_old;
        }
#endif
}

DEFINE_UDS_FLUX(uds_flux, f, t, i)
{
        cell_t c0, c1 = -1;
        Thread *t0, *t1 = NULL;

        real  flux = 0.0;
        real Area[ND_ND], ds, es[ND_ND], A_by_es, dr0[ND_ND], dr1[ND_ND];
        c0 = F_C0(f, t);
        t0 = F_C0_THREAD(f, t);

        /* If face lies at domain boundary, use face values; */
        /* If face lies IN the domain, use average of adjacent cells. */
        if (BOUNDARY_FACE_THREAD_P(t)) /*Most face values will be available*/
        {
                flux = F_FLUX(f, t) * cv(c0, t0); /* flux through Face */
        }
        else
        {
                c1 = F_C1(f, t);   /* Get cell on other side of face */
                t1 = F_C1_THREAD(f, t);
                flux = F_FLUX(f, t) * (cv(c0, t0) + cv(c1, t1)) / 2.0; /* Average flux through face */
        }

        return flux;
}

DEFINE_SOURCE(source_sensible_heat, c, t, dS, eqn)
{
        real dt, source;
        dt = CURRENT_TIMESTEP;
        source = -rho_m1(c, t)*(e_sen(c, t) - e_sen_m1(c, t)) / dt;
        dS[eqn] = -rho_m1(c, t) * cv_m1(c, t) / dt;
        return source;
}
DEFINE_SOURCE(source_latent_heat, c, t, dS, eqn)
{
        real dt, source;
        dt = CURRENT_TIMESTEP;
        source = rho_m1(c, t) * delta_e * (eof_m1(c, t) - eof(c, t)) / dt;
        dS[eqn] = 0;
        return source;
}

DEFINE_SOURCE(source_bouyancy, c, t, dS, eqn)
{
        real source;
        source = 9.81 * (rho(c, t) - rho0);
        dS[eqn] = 0;
        return source;
}
DEFINE_SOURCE(cancel_x, c, t, dS, eqn)
{
        real source;
        if (C_interface(c, t))
        {
                real power = pow(10, C_power(c, t));
                source = -power * C_U(c, t);
                dS[eqn] = -power;
        }
        else
        {
                source = 0;
                dS[eqn] = 0;
        }
        return source;
}
DEFINE_SOURCE(cancel_y, c, t, dS, eqn)
{
        real source;
        if (C_interface(c, t))
        {
                real power = pow(10, C_power(c, t));
                source = -power * C_V(c, t);
                dS[eqn] = -power;
        }
        else
        {
                source = 0;
                dS[eqn] = 0;
        }
        return source;
}

DEFINE_PROPERTY(property_density, c, t)
{
        return 71.6553;
}
DEFINE_PROPERTY(property_viscosity, c, t)
{
        return mu(c, t);
}
DEFINE_DIFFUSIVITY(property_diffusivity, c, t, i)
{
        return k(c, t);
}

/*******************************************
End FLUENT Iteration Loop (Covergence Check)
*******************************************/

DEFINE_EXECUTE_AT_END(hook_execute_at_end)
{
        iter = 0;
        int i, ii;
#if !RP_HOST
        if (CURRENT_TIME > 0)
        {
                for (i = i_start;i <= i_end; i++)
                {
                        HR_wall[i] = HF_wall[i] * A_wall[i];
                        HI_wall[i] += HR_wall[i] * CURRENT_TIMESTEP;
```

```
        }
}
Domain *d = Get_Domain(1);
Thread *t = Lookup_Thread(d, fluid);
cell_t c;
Node *node;
real x[ND_ND];

/* Reconstruct Interface */
if (N_TIME % 1 == 0)
{
        real y_test, y_old, y_new, dVdy, dy = 0.00000001;
        int j;

        y_old = interf_new;
        Message0("\nOld Interface: %#7.6g\n", interf_new);

        int iter = 1;
        real tol = 1e-9;
        resid = 1.0;
        Message0(" i     y_test    vol_test       dVdy       y_old       y_new       resid\n");
        while (resid > tol && iter < 20)
        {
                Message0(" %d ", iter);
                double vol_test[3] = { 0 };
                for (j = 0; j <= 2; j++)
                {
                        y_test = y_old + (j - 1)*dy;
                        Message0(" %#10.8g ", y_test);

                        /* Recalculate EOF */
                        begin_c_loop(c, t)
                        {
                                C_CENTROID(x, c, t);
                                real max = -1e6;
                                real min = 1e6;
                                c_node_loop(c, t, i)
                                {
                                        node = C_NODE(c, t, i);
                                        if (NODE_X(node) > max) max = NODE_X(node);
                                        if (NODE_X(node) < min) min = NODE_X(node);
                                }
                                if (max > y_test && min > y_test) eofold(c, t) = 0;
                                if (max > y_test && min < y_test) eofold(c, t) = (y_test - min) / (max - min);
                                if (max < y_test && min < y_test) eofold(c, t) = 1;
                        }
                        end_c_loop(c, t)

                        /* Calculate new liquid volume */
                        begin_c_loop_int(c, t)
                        {
                                if (eofold(c, t) != 1) vol_test[j] += vol(c, t)*(1 - eofold(c, t));
                        }
                        end_c_loop_int(c, t)

                        vol_test[j] = PRF_GRSUM1(vol_test[j]);
                        Message0(" %10.4e ", vol_test[j]);
                        if (j < 2) Message0("\n   ");
                }

                dVdy = (vol_test[2] - vol_test[0]) / (2 * dy);
                Message0(" %+10.3e ", dVdy);
                y_new = y_old - (vol_test[1] - vol_liq_sum) / dVdy;
                Message0(" %#10.8g  %#10.8g ", y_old, y_new);

                resid = fabs(y_new - y_old) / y_old;
                Message0(" %10.4e \n", resid);
                y_old = y_new;
                iter++;
        }
        if (resid > tol) Message0("Interface reconstruction did not converge!\n");
        interf_new = y_new;
        Message0("New Interface: %#7.6g  %10.4e\n", interf_new, resid);

        /* Reconstruct Interface */
        begin_c_loop(c, t)
        {
                C_CENTROID(x, c, t);
                real max = -1e6;
                real min = 1e6;
                c_node_loop(c, t, i)
                {
                        node = C_NODE(c, t, i);
                        if (NODE_X(node) > max) max = NODE_X(node);
                        if (NODE_X(node) < min) min = NODE_X(node);
                }
                if (max > y_test && min > y_test) eof(c, t) = 0;
                if (max > y_test && min < y_test) eof(c, t) = (interf_new - min) / (max - min);
                if (max < y_test && min < y_test) eof(c, t) = 1;
        }
        end_c_loop(c, t)
}

/* Update source power */
if (N_TIME % 1 == 0)
{
        double* dfdtsum = (double*)calloc(col, sizeof(double));
        double* eofsum = (double*)calloc(col, sizeof(double));
        double* eofhigh = (double*)calloc(col, sizeof(double));
        double* eoflow = (double*)calloc(col, sizeof(double));
        for (i = 0;i < col;i++) eoflow[i] = 2;
        int* count_total = (int*)calloc(col, sizeof(int));
        double* pwrold = (double*)calloc(col, sizeof(double));
        double* pwrnew = (double*)calloc(col, sizeof(double));
        double* centroid = (double*)calloc(col, sizeof(double));
        real dt = CURRENT_TIMESTEP;

        /* Sum interface cells by column */
        begin_c_loop_int(c, t)
        {
                if (C_search(c, t))
                {
                        /* Index column */
                        C_CENTROID(x, c, t);
                        for (i = 0;i < col;i++)
                                if (x[0] < x_start + cell_size * (i + 1))
```

```
                        {
                                ii = i;
                                break;
                        }

                /* Sum column phase change data */
                dfdtsum[ii] += (eof(c, t) - eof_m1(c, t)) / dt;
                eofsum[ii] += eof(c, t);
                if (eof(c, t) > eofhigh[ii]) eofhigh[ii] = eof(c, t);
                if (eof(c, t) < eoflow[ii]) eoflow[ii] = eof(c, t);
                pwrold[ii] += C_power(c, t);
                centroid[ii] += x[0];
                count_total[ii]++;
        }
    }
    end_c_loop_int(c, t)

    /* Sum or sync across processing nodes */
    for (i = 0;i < col; i++)
    {
        dfdtsum[i] = PRF_GRSUM1(dfdtsum[i]);
        eofsum[i] = PRF_GRSUM1(eofsum[i]);
        eofhigh[i] = PRF_GRHIGH1(eofhigh[i]);
        eoflow[i] = PRF_GRLOW1(eoflow[i]);
        pwrold[i] = PRF_GRSUM1(pwrold[i]);
        centroid[i] = PRF_GRSUM1(centroid[i]);
        count_total[i] = PRF_GISUM1(count_total[i]);
    }

    /* Use column averages to determine where to put power */
    begin_c_loop(c, t)
    {
        if (C_search(c, t))
        {
            /* Index column */
            C_CENTROID(x, c, t);
            for (i = 0;i < col;i++)
                if (x[0] < x_start + cell_size * (i + 1))
                {
                    ii = i;
                    break;
                }

            /* Give power to cells when adjacent column is about to completely
condense/evaporate */
            real buffer = 0.15 - 100 * cell_size;
            if (ii < col)
                if (dfdtsum[ii + 1] < 0) /* condensation */
                    if (eofsum[ii + 1] / count_total[ii + 1] < buffer)
                        C_power(c, t) += dt * (1 + 10);
            if (ii >= 1)
                if (dfdtsum[ii - 1] > 0) /* evaporation */
                    if (eofsum[ii - 1] / count_total[ii - 1] > 1.0 - buffer)
                        C_power(c, t) += dt * (1 + 10);

            /* Give power to mixed phase cells */
            if (0 < eofsum[ii] / count_total[ii] && eofsum[ii] / count_total[ii] < 1) C_power(c,
t) += 1 * dt;

            /* Remove power from pure phase cells */
            if (eofhigh[ii] == 0 || eoflow[ii] == 1) C_power(c, t) -= 1 * dt;

            /* Give power to cells when phase front is directly in between two columns */
            if (ii < col)
                if (eofhigh[ii+1] == 0 && eoflow[ii] == 1)
                    C_power(c, t) += 1 * dt;

            /* enforce limits */
            if (C_power(c, t) < -4) C_power(c, t) = -4;
            if (C_power(c, t) > 16) C_power(c, t) = 16;

            /* update interface mark */
            if (C_power(c, t) > -4) C_interface(c, t) = 1;
            else C_interface(c, t) = 0;
        }
    }
    end_c_loop(c, t)

    /* Calculate change for monitoring purposes */
    PRF_GSYNC();
    begin_c_loop_int(c, t)
    {
        if (C_search(c, t))
        {
            C_CENTROID(x, c, t);
            for (i = 0;i < col;i++)
                if (x[0] < x_start + cell_size * (i + 1))
                {
                    ii = i;
                    break;
                }
            pwrnew[ii] += C_power(c, t);
        }
    }
    end_c_loop_int(c, t)
    for (i = 0;i < col; i++) pwrnew[i] = PRF_GRSUM1(pwrnew[i]);

    /* Determine which columns to show */
    int i0 = 100;
    int iN = 0;
    for (i = 0;i < col;i++)
    {
        int i0_temp, iN_temp;

        /* Column has mixed phase */
        if (0 < eofsum[i] / count_total[i] && eofsum[i] / count_total[i] < 1)
        {
            i0_temp = i - 2;
            if (i0_temp < i0) i0 = i0_temp;
            iN_temp = i + 2;
            if (iN_temp > iN) iN = iN_temp;
        }

        /* Column is between cells */
        if (i < col)
            if (eofhigh[i + 1] == 0 && eoflow[i] == 1)
```

```
                {
                        i0_temp = i - 2;
                        if (i0_temp < i0) i0 = i0_temp;
                        iN_temp = i + 1;
                        if (iN_temp > iN) iN = iN_temp;
                }

        if (i0 < 0) i0 = 0;
        if (iN > col - 1) iN = col - 1;
    }

    /* Report Values */
    Message0("\nInterface Report:\n");
    Message0("   Column: ");
    for (i = i0;i <= iN; i++) Message0("       %2d ", i);
    Message0("\n");
    Message0("  Cell Ct: ");
    for (i = i0;i <= iN; i++) Message0("     %4d ", count_total[i]);
    Message0("\n");
    Message0(" Centroid: ");
    for (i = i0;i <= iN; i++) Message0("  %#7.5g ", centroid[i] / count_total[i]);
    Message0("\n");
    Message0("    Power: ");
    for (i = i0;i <= iN; i++) Message0(" %#+8.6g ", pwrnew[i] / count_total[i]);
    Message0("\n");
    Message0("  Pwr Chg: ");
    for (i = i0;i <= iN; i++) Message0(" %#+.5f ", (pwrnew[i] - pwrold[i]) / count_total[i]);
    Message0("\n");
    Message0("  EOF Avg: ");
    for (i = i0;i <= iN; i++) Message0("  %#.5f ", eofsum[i] / count_total[i]);
    Message0("\n");
    Message0("  EOF Min: ");
    for (i = i0;i <= iN; i++) Message0("  %#.5f ", eoflow[i]);
    Message0("\n");
    Message0("  EOF Max: ");
    for (i = i0;i <= iN; i++) Message0("  %#.5f ", eofhigh[i]);
    Message0("\n");
    Message0("    df/dt: ");
    for (i = i0;i <= iN; i++) Message0(" %#+8.1e ", dfdtsum[i] / count_total[i]);
    Message0("\n");

    free(dfdtsum);
    free(eofsum);
    free(eofhigh);
    free(eoflow);
    free(count_total);
    free(pwrold);
    free(pwrnew);
    free(centroid);
}

Message0("\n");

/* Store for next time step*/
begin_c_loop(c, t)
{
    eof_m1(c, t) = eof(c, t);
    e_m1(c, t) = e(c, t);
            e_sen_m1(c, t) = e_sen(c, t);
            cv_m1(c, t) = cv(c, t);
            rho_m1(c, t) = rho(c, t);
        }
        end_c_loop(c, t)
        PRF_GSYNC();
#endif
}

DEFINE_REPORT_DEFINITION_FN(timestep_ender)
{
    node_to_host_int_1(iter);
    return iter;
}
DEFINE_REPORT_DEFINITION_FN(report_mass_liq)
{
    node_to_host_real_1(mass_liq);
    return mass_liq;
}
DEFINE_REPORT_DEFINITION_FN(report_mass_vap)
{
    node_to_host_real_1(mass_vap);
    return mass_vap;
}
DEFINE_REPORT_DEFINITION_FN(report_real_mass_tot)
{
    node_to_host_real_1(real_mass_tot);
    return real_mass_tot;
}
DEFINE_REPORT_DEFINITION_FN(report_real_mass_vap)
{
    node_to_host_real_1(real_mass_vap);
    return real_mass_vap;
}
DEFINE_REPORT_DEFINITION_FN(report_real_mass_liq)
{
    node_to_host_real_1(real_mass_liq);
    return real_mass_liq;
}
DEFINE_REPORT_DEFINITION_FN(report_TVbulk)
{
    node_to_host_real_1(T_vap_bulk);
    return T_vap_bulk;
}
DEFINE_REPORT_DEFINITION_FN(report_rhoVbulk)
{
    node_to_host_real_1(rho_vap_bulk);
    return rho_vap_bulk;
}
DEFINE_REPORT_DEFINITION_FN(report_ELF)
{
    node_to_host_real_1(elf);
    return elf;
}
DEFINE_REPORT_DEFINITION_FN(report_Psat)
{
    node_to_host_real_1(P_sat);
    return P_sat;
```

```
}
DEFINE_REPORT_DEFINITION_FN(report_Tsat)
{
	node_to_host_real_1(T_sat);
	return T_sat;
}
DEFINE_REPORT_DEFINITION_FN(report_hf_dry_wall)
{
	node_to_host_real_1(HF_dry);
	return HF_dry;
}
DEFINE_REPORT_DEFINITION_FN(report_hf_wet_wall)
{
	node_to_host_real_1(HF_wet);
	return HF_wet;
}
DEFINE_REPORT_DEFINITION_FN(report_hr_dry_wall)
{
	node_to_host_real_1(HR_dry);
	return HR_dry;
}
DEFINE_REPORT_DEFINITION_FN(report_hr_wet_wall)
{
	node_to_host_real_1(HR_wet);
	return HR_wet;
}
DEFINE_REPORT_DEFINITION_FN(report_hi_dry_wall)
{
	node_to_host_real_1(HI_dry);
	return HI_dry;
}
DEFINE_REPORT_DEFINITION_FN(report_hi_wet_wall)
{
	node_to_host_real_1(HI_wet);
	return HI_wet;
}
DEFINE_REPORT_DEFINITION_FN(report_rho0)
{
	node_to_host_real_1(rho0);
	return rho0;
}
DEFINE_REPORT_DEFINITION_FN(report_interf)
{
	node_to_host_real_1(interf_new);
	return interf_new;
}
```

APPENDIX K SOURCE CODE FOR PARAHYDROGEN PROPERTY TABLES

```
/************************************************************
LIST OF FUNCTIONS IN THIS SOURCE CODE

Function      Method                Code Source      Data/Eqn Source
kfTsat        Interpolation         Winter 2014      NIST
mufTsat       Interpolation         Winter 2014      NIST
compressed    Interpolation         Winter 2014      NIST
superheat     Interpolation         Winter 2014      NIST
PfTR          Curve Fit             Clark 2002       Reynolds 1979
EfTR          Integration           Clark 2002       Reynolds 1979
CvfT          Curve Fit             Clark 2002       Reynolds 1979
RfTP          Root Find             Original         Reynolds 1979
PsatfTsat     Curve Fit             Clark 2002       Reynolds 1979
TsatfPsat     Root Find             Original         Reynolds 1979
RVfTsat       Curve Fit             Original         Younglove 1982
RLfTsat       Curve Fit             Original         Younglove 1982
RfTsat        Function Hub          Original
prop          Function Hub          Original
meanof        Interpolation         Winter 2014
************************************************************/

#include "udf.h"

/* Index counter */
int i;

/* Sizes of property data arrays */
#define imax 31
#define jmax 15
#define kmax 1

/* Constants */
double Tcrit = 32.938;
double Pcrit = 1.28377e6;
double Tmin = 19.505;
double Pmin = 8.0e4;
double Pmax = 1.0e6;
double Tempmax = 400.0;

/* Temperature and Pressure Axes */
double Pres[jmax] = {
        8.00000e4,  9.00000e4,  1.00000e5,  1.20000e5,  1.50000e5,  2.00000e5,  2.50000e5,  3.00000e5,  4.00000e5,  5.00000e5,  6.00000e5,  7.00000e5,  8.00000e5,  9.00000e5,  1.00000e6
};
double Tsat[jmax] = {
        19.5047,    19.8835,    20.2330,    20.8627,    21.6790,    22.8117,    23.7582,    24.5791,    25.9686,    27.1317,    28.1411,    29.0380,    29.8477,    30.5873,    31.2684
};
double Tem[imax] = {
        19.5047,
        19.8835,
        20.2330,
        20.8627,
        21.6790,
        22.8117,
        23.7582,
```

```
        24.5791,
        25.9686,
        27.1317,
        28.1411,
        29.0380,
        29.8477,
        30.5873,
        31.2684,
        35.0000,
        50.0000,
        75.0000,
        100.000,
        125.000,
        150.000,
        175.000,
        200.000,
        225.000,
        250.000,
        275.000,
        300.000,
        325.000,
        350.000,
        375.000,
        400.000
};

/* Liquid Property Data Tables */
double kL[kmax][jmax] = {
        0.102133,  0.102810,  0.103347,  0.104112,  0.104747,  0.105026,  0.104773,  0.104220,  0.102606,  0.100594,  0.0983187,  0.0958252,  0.0931118,  0.0901396,  0.0868138
};
double muL[kmax][jmax] = {
        1.41764e-5, 1.37377e-5, 1.33524e-5, 1.27007e-5, 1.19266e-5, 1.09614e-5, 1.02334e-5, 9.64830e-6, 8.73485e-6, 8.02486e-6, 7.43443e-6, 6.91914e-6, 6.45129e-6, 6.01018e-6, 5.57593e-6
};

/* Vapor Property Data Tables */
double kV[imax][jmax] = {
        0.0161710,  0.0165783,  0.0169594,  0.0176605,  0.0186006,  0.0199757,  0.0212025,  0.0223380,  0.0244555,  0.0264854,  0.0285209,  0.0306383,  0.0329208,  0.0354818,  0.0385091,
        0.0164838,  0.0165783,  0.0169594,  0.0176605,  0.0186006,  0.0199757,  0.0212025,  0.0223380,  0.0244555,  0.0264854,  0.0285209,  0.0306383,  0.0329208,  0.0354818,  0.0385091,
        0.0167720,  0.0168644,  0.0169594,  0.0176605,  0.0186006,  0.0199757,  0.0212025,  0.0223380,  0.0244555,  0.0264854,  0.0285209,  0.0306383,  0.0329208,  0.0354818,  0.0385091,
        0.0172903,  0.0173791,  0.0174703,  0.0176605,  0.0186006,  0.0199757,  0.0212025,  0.0223380,  0.0244555,  0.0264854,  0.0285209,  0.0306383,  0.0329208,  0.0354818,  0.0385091,
        0.0179600,  0.0180447,  0.0181313,  0.0183112,  0.0186006,  0.0199757,  0.0212025,  0.0223380,  0.0244555,  0.0264854,  0.0285209,  0.0306383,  0.0329208,  0.0354818,  0.0385091,
        0.0188845,  0.0189642,  0.0190455,  0.0192133,  0.0194800,  0.0199757,  0.0212025,  0.0223380,  0.0244555,  0.0264854,  0.0285209,  0.0306383,  0.0329208,  0.0354818,  0.0385091,
        0.0196521,  0.0197282,  0.0198055,  0.0199646,  0.0202158,  0.0206755,  0.0212025,  0.0223380,  0.0244555,  0.0264854,  0.0285209,  0.0306383,  0.0329208,  0.0354818,  0.0385091,
        0.0203137,  0.0203869,  0.0204612,  0.0206136,  0.0208529,  0.0212867,  0.0217754,  0.0223380,  0.0244555,  0.0264854,  0.0285209,  0.0306383,  0.0329208,  0.0354818,  0.0385091,
        0.0214235,  0.0214924,  0.0215621,  0.0217046,  0.0219268,  0.0223241,  0.0227625,  0.0232525,  0.0244555,  0.0264854,  0.0285209,  0.0306383,  0.0329208,  0.0354818,  0.0385091,
        0.0223418,  0.0224075,  0.0224738,  0.0226090,  0.0228189,  0.0231911,  0.0235966,  0.0240423,  0.0250947,  0.0264854,  0.0285209,  0.0306383,  0.0329208,  0.0354818,  0.0385091,
        0.0231301,  0.0231932,  0.0232569,  0.0233864,  0.0235869,  0.0239401,  0.0243216,  0.0247364,  0.0256934,  0.0268931,  0.0285209,  0.0306383,  0.0329208,  0.0354818,  0.0385091,
        0.0238234,  0.0238844,  0.0239460,  0.0240709,  0.0242637,  0.0246019,  0.0249647,  0.0253563,  0.0262456,  0.0273254,  0.0287024,  0.0306383,  0.0329208,  0.0354818,  0.0385091,
        0.0244435,  0.0245027,  0.0245624,  0.0246834,  0.0248698,  0.0251957,  0.0255435,  0.0259167,  0.0267545,  0.0277500,  0.0289738,  0.0305684,  0.0329208,  0.0354818,  0.0385091,
        0.0250048,  0.0250625,  0.0251206,  0.0252383,  0.0254193,  0.0257347,  0.0260700,  0.0264281,  0.0272250,  0.0281572,  0.0292751,  0.0306699,  0.0325376,  0.0354818,  0.0385091,
        0.0255175,  0.0255739,  0.0256306,  0.0257455,  0.0259217,  0.0262281,  0.0265528,  0.0268982,  0.0276615,  0.0284492,  0.0295831,  0.0308436,  0.0324449,  0.0346634,  0.0385091,
        0.0282549,  0.0283050,  0.0283552,  0.0284564,  0.0286107,  0.0288762,  0.0291536,  0.0294442,  0.0300697,  0.0307625,  0.0315341,  0.0323998,  0.0333802,  0.0345040,  0.0358132,
        0.0385075,  0.0385419,  0.0385763,  0.0386450,  0.0387483,  0.0389223,  0.0390994,  0.0392803,  0.0396543,  0.0400458,  0.0404556,  0.0408844,  0.0413328,  0.0418013,  0.0422907,
        0.0571346,  0.0571596,  0.0571844,  0.0572336,  0.0573070,  0.0574287,  0.0575505,  0.0576727,  0.0579196,  0.0581703,  0.0584252,  0.0586845,  0.0589484,  0.0592167,  0.0594895,
        0.0803860,  0.0804102,  0.0804342,  0.0804819,  0.0805527,  0.0806695,  0.0807854,  0.0809008,  0.0811307,  0.0813603,  0.0815899,  0.0818198,  0.0820503,  0.0822813,  0.0825131,
        0.106144,  0.106164,  0.106183,  0.106222,  0.106279,  0.106373,  0.106466,  0.106558,  0.106741,  0.106923,  0.107105,  0.107287,  0.107468,  0.107649,  0.107831,
        0.127730,  0.127747,  0.127764,  0.127796,  0.127845,  0.127925,  0.128004,  0.128082,  0.128236,  0.128390,  0.128542,  0.128694,  0.128846,  0.128998,  0.129150,
        0.143683,  0.143697,  0.143712,  0.143741,  0.143783,  0.143853,  0.143922,  0.143990,  0.144124,  0.144257,  0.144389,  0.144521,  0.144652,  0.144783,  0.144914,
```

```
	0.155609,	0.155623,	0.155636,	0.155662,	0.155700,	0.155762,	0.155823,	0.155883,	0.156003,	0.156120,	0.156238,	0.156354,	0.156470,	0.156586,	0.156701,
	0.165155,	0.165168,	0.165180,	0.165203,	0.165238,	0.165294,	0.165349,	0.165404,	0.165511,	0.165617,	0.165723,	0.165827,	0.165932,	0.166035,	0.166139,
	0.174163,	0.174174,	0.174185,	0.174207,	0.174239,	0.174290,	0.174341,	0.174391,	0.174489,	0.174586,	0.174682,	0.174777,	0.174872,	0.174966,	0.175061,
	0.183078,	0.183088,	0.183098,	0.183118,	0.183148,	0.183196,	0.183243,	0.183289,	0.183380,	0.183469,	0.183557,	0.183645,	0.183732,	0.183819,	0.183905,
	0.192243,	0.192253,	0.192263,	0.192282,	0.192309,	0.192354,	0.192398,	0.192441,	0.192525,	0.192608,	0.192690,	0.192772,	0.192853,	0.192933,	0.193013,
	0.201905,	0.201914,	0.201923,	0.201941,	0.201967,	0.202009,	0.202050,	0.202091,	0.202170,	0.202247,	0.202324,	0.202400,	0.202476,	0.202551,	0.202625,
	0.212205,	0.212214,	0.212223,	0.212240,	0.212265,	0.212305,	0.212344,	0.212382,	0.212456,	0.212529,	0.212602,	0.212673,	0.212744,	0.212815,	0.212885,
	0.222502,	0.222510,	0.222518,	0.222535,	0.222558,	0.222596,	0.222633,	0.222669,	0.222740,	0.222809,	0.222878,	0.222945,	0.223012,	0.223079,	0.223145,
	0.233039,	0.233047,	0.233055,	0.233071,	0.233094,	0.233130,	0.233165,	0.233200,	0.233267,	0.233333,	0.233398,	0.233462,	0.233525,	0.233589,	0.233651
};
double muV[imax][jmax] = {
	1.02671E-6, 1.04978E-6, 1.07117E-6, 1.11007E-6, 1.16137E-6, 1.23470E-6, 1.29841E-6, 1.35591E-6, 1.45944E-6, 1.55453E-6, 1.64667E-6, 1.74050E-6, 1.84132E-6, 1.95674E-6, 2.09967E-6,
	1.04657E-6, 1.04978E-6, 1.07117E-6, 1.11007E-6, 1.16137E-6, 1.23470E-6, 1.29841E-6, 1.35591E-6, 1.45944E-6, 1.55453E-6, 1.64667E-6, 1.74050E-6, 1.84132E-6, 1.95674E-6, 2.09967E-6,
	1.06480E-6, 1.06794E-6, 1.07117E-6, 1.11007E-6, 1.16137E-6, 1.23470E-6, 1.29841E-6, 1.35591E-6, 1.45944E-6, 1.55453E-6, 1.64667E-6, 1.74050E-6, 1.84132E-6, 1.95674E-6, 2.09967E-6,
	1.09747E-6, 1.10051E-6, 1.10361E-6, 1.11007E-6, 1.16137E-6, 1.23470E-6, 1.29841E-6, 1.35591E-6, 1.45944E-6, 1.55453E-6, 1.64667E-6, 1.74050E-6, 1.84132E-6, 1.95674E-6, 2.09967E-6,
	1.13957E-6, 1.14248E-6, 1.14545E-6, 1.14849E-6, 1.16137E-6, 1.23470E-6, 1.29841E-6, 1.35591E-6, 1.45944E-6, 1.55453E-6, 1.64667E-6, 1.74050E-6, 1.84132E-6, 1.95674E-6, 2.09967E-6,
	1.19765E-6, 1.20041E-6, 1.20322E-6, 1.20900E-6, 1.21810E-6, 1.23470E-6, 1.29841E-6, 1.35591E-6, 1.45944E-6, 1.55453E-6, 1.64667E-6, 1.74050E-6, 1.84132E-6, 1.95674E-6, 2.09967E-6,
	1.24597E-6, 1.24862E-6, 1.25131E-6, 1.25683E-6, 1.26545E-6, 1.28098E-6, 1.29841E-6, 1.35591E-6, 1.45944E-6, 1.55453E-6, 1.64667E-6, 1.74050E-6, 1.84132E-6, 1.95674E-6, 2.09967E-6,
	1.28774E-6, 1.29030E-6, 1.29290E-6, 1.29822E-6, 1.30649E-6, 1.32124E-6, 1.33752E-6, 1.35591E-6, 1.45944E-6, 1.55453E-6, 1.64667E-6, 1.74050E-6, 1.84132E-6, 1.95674E-6, 2.09967E-6,
	1.35816E-6, 1.36059E-6, 1.36305E-6, 1.36807E-6, 1.37583E-6, 1.38950E-6, 1.40428E-6, 1.42048E-6, 1.45944E-6, 1.55453E-6, 1.64667E-6, 1.74050E-6, 1.84132E-6, 1.95674E-6, 2.09967E-6,
	1.41680E-6, 1.41914E-6, 1.42150E-6, 1.42630E-6, 1.43369E-6, 1.44662E-6, 1.46042E-6, 1.47531E-6, 1.50967E-6, 1.55453E-6, 1.64667E-6, 1.74050E-6, 1.84132E-6, 1.95674E-6, 2.09967E-6,
	1.46744E-6, 1.46970E-6, 1.47198E-6, 1.47661E-6, 1.48372E-6, 1.49608E-6, 1.50918E-6, 1.52316E-6, 1.55464E-6, 1.59343E-6, 1.64667E-6, 1.74050E-6, 1.84132E-6, 1.95674E-6, 2.09967E-6,
	1.51222E-6, 1.51441E-6, 1.51663E-6, 1.52111E-6, 1.52800E-6, 1.53991E-6, 1.55246E-6, 1.56575E-6, 1.59522E-6, 1.63027E-6, 1.67502E-6, 1.74050E-6, 1.84132E-6, 1.95674E-6, 2.09967E-6,
	1.55244E-6, 1.55458E-6, 1.55674E-6, 1.56111E-6, 1.56780E-6, 1.57935E-6, 1.59145E-6, 1.60420E-6, 1.63215E-6, 1.66461E-6, 1.70432E-6, 1.75729E-6, 1.84132E-6, 1.95674E-6, 2.09967E-6,
	1.58899E-6, 1.59109E-6, 1.59320E-6, 1.59747E-6, 1.60400E-6, 1.61523E-6, 1.62697E-6, 1.63928E-6, 1.66602E-6, 1.69655E-6, 1.73285E-6, 1.77876E-6, 1.84331E-6, 1.95674E-6, 2.09967E-6,
	1.62250E-6, 1.62455E-6, 1.62662E-6, 1.63080E-6, 1.63719E-6, 1.64816E-6, 1.65958E-6, 1.67152E-6, 1.69727E-6, 1.72630E-6, 1.76011E-6, 1.80140E-6, 1.85572E-6, 1.93690E-6, 2.09967E-6,
	1.80301E-6, 1.80487E-6, 1.80674E-6, 1.81050E-6, 1.81622E-6, 1.82596E-6, 1.83599E-6, 1.84633E-6, 1.86807E-6, 1.89151E-6, 1.91712E-6, 1.94557E-6, 1.97785E-6, 2.01544E-6, 2.06071E-6,
	2.46897E-6, 2.47032E-6, 2.47168E-6, 2.47440E-6, 2.47850E-6, 2.48539E-6, 2.49235E-6, 2.49939E-6, 2.51369E-6, 2.52834E-6, 2.54335E-6, 2.55878E-6, 2.57467E-6, 2.59106E-6, 2.60802E-6,
	3.39799E-6, 3.39893E-6, 3.39987E-6, 3.40175E-6, 3.40457E-6, 3.40929E-6, 3.41402E-6, 3.41878E-6, 3.42830E-6, 3.43790E-6, 3.44757E-6, 3.45730E-6, 3.46711E-6, 3.47700E-6, 3.48698E-6,
	4.18813E-6, 4.18885E-6, 4.18957E-6, 4.19101E-6, 4.19317E-6, 4.19678E-6, 4.20038E-6, 4.20400E-6, 4.21123E-6, 4.21848E-6, 4.22574E-6, 4.23302E-6, 4.24033E-6, 4.24765E-6, 4.25499E-6,
	4.89922E-6, 4.89980E-6, 4.90037E-6, 4.90152E-6, 4.90325E-6, 4.90613E-6, 4.90900E-6, 4.91188E-6, 4.91764E-6, 4.92340E-6, 4.92916E-6, 4.93492E-6, 4.94069E-6, 4.94647E-6, 4.95225E-6,
	5.55968E-6, 5.56016E-6, 5.56064E-6, 5.56160E-6, 5.56303E-6, 5.56543E-6, 5.56782E-6, 5.57022E-6, 5.57500E-6, 5.57979E-6, 5.58457E-6, 5.58935E-6, 5.59414E-6, 5.59892E-6, 5.60371E-6,
	6.18366E-6, 6.18407E-6, 6.18448E-6, 6.18530E-6, 6.18653E-6, 6.18859E-6, 6.19064E-6, 6.19269E-6, 6.19679E-6, 6.20088E-6, 6.20498E-6, 6.20907E-6, 6.21316E-6, 6.21725E-6, 6.22133E-6,
	6.77904E-6, 6.77939E-6, 6.77975E-6, 6.78047E-6, 6.78155E-6, 6.78335E-6, 6.78514E-6, 6.78693E-6, 6.79052E-6, 6.79410E-6, 6.79768E-6, 6.80126E-6, 6.80483E-6, 6.80841E-6, 6.81198E-6,
	7.35068E-6, 7.35100E-6, 7.35132E-6, 7.35196E-6, 7.35292E-6, 7.35451E-6, 7.35611E-6, 7.35770E-6, 7.36089E-6, 7.36407E-6, 7.36725E-6, 7.37043E-6, 7.37361E-6, 7.37678E-6, 7.37996E-6,
	7.90193E-6, 7.90222E-6, 7.90251E-6, 7.90308E-6, 7.90394E-6, 7.90538E-6, 7.90682E-6, 7.90825E-6, 7.91112E-6, 7.91398E-6, 7.91685E-6, 7.91971E-6, 7.92257E-6, 7.92542E-6, 7.92828E-6,
	8.43526E-6, 8.43552E-6, 8.43578E-6, 8.43630E-6, 8.43709E-6, 8.43839E-6, 8.43970E-6, 8.44100E-6, 8.44361E-6, 8.44621E-6, 8.44882E-6, 8.45142E-6, 8.45402E-6, 8.45661E-6, 8.45921E-6,
	8.95262E-6, 8.95286E-6, 8.95309E-6, 8.95357E-6, 8.95429E-6, 8.95549E-6, 8.95669E-6, 8.95788E-6, 8.96027E-6, 8.96266E-6, 8.96505E-6, 8.96743E-6, 8.96981E-6, 8.97220E-6, 8.97457E-6,
	9.45562E-6, 9.45585E-6, 9.45607E-6, 9.45651E-6, 9.45717E-6, 9.45828E-6, 9.45938E-6, 9.46049E-6, 9.46269E-6, 9.46490E-6, 9.46710E-6, 9.46930E-6, 9.47150E-6, 9.47370E-6, 9.47590E-6,
	9.94566E-6, 9.94586E-6, 9.94607E-6, 9.94648E-6, 9.94710E-6, 9.94812E-6, 9.94915E-6, 9.95017E-6, 9.95222E-6, 9.95427E-6, 9.95632E-6, 9.95836E-6, 9.96041E-6, 9.96245E-6, 9.96449E-6,
	1.04239E-5, 1.04241E-5, 1.04243E-5, 1.04247E-5, 1.04253E-5, 1.04262E-5, 1.04272E-5, 1.04281E-5, 1.04300E-5, 1.04320E-5, 1.04339E-5, 1.04369E-5, 1.04378E-5, 1.04396E-5, 1.04415E-5,
	1.08914E-5, 1.08916E-5, 1.08918E-5, 1.08922E-5, 1.08927E-5, 1.08936E-5, 1.08945E-5, 1.08954E-5, 1.08972E-5, 1.08990E-5, 1.09008E-5, 1.09026E-5, 1.09043E-5, 1.09061E-5, 1.09079E-5
};

/* Saturated Thermal Conductivity
	Based on data tables recorded from NIST
	1 Input: Temperature K
	2 Outputs: Saturated Liquid Thermal Conductivity W/m-K, Saturated Vapor Thermal Conductivity W/m-K*/
void kfTsat(double fTsat, double *ykL, double *ykV)
{
	double fxubound, yubound, fxlbound, ylbound;
	int j, extrap = 0;

	if (fTsat >= Tcrit) Error("Calculted bulk temperature above critical temperature in Function kfTsat");

	/*For saturated vapor-Tsat[j] corresponds to P[j]*/
	for (j = 1;j < jmax;j++)
	{
		if (Tsat[j] > fTsat)
		{
			fxubound = Tsat[j];
			yubound = kV[j][j];
			fxlbound = Tsat[j - 1];
			ylbound = kV[j - 1][j - 1];
			extrap = 0;
			break;
		}
		else
		{
			extrap = 1;
		}
```

```
	}

	if (extrap != 0) /*need to extropolate data based on table of know values*/
	{
		fxubound = Tsat[jmax - 1];
		yubound = kV[jmax - 1][jmax - 1];
		fxlbound = Tsat[jmax - 2];
		ylbound = kV[jmax - 2][jmax - 2];
	}

	*ykV = ylbound + (yubound - ylbound) / (fxubound - fxlbound)*(fTsat - fxlbound);

	/*For saturated liquid-Tsat[j] corresponds to P[j]*/
	for (j = 1;j < jmax;j++)
	{
		if (Tsat[j] > fTsat)
		{
			fxubound = Tsat[j];
			yubound = kL[0][j];
			fxlbound = Tsat[j - 1];
			ylbound = kL[0][j - 1];
			extrap = 0;
			break;
		}
		else
		{
			extrap = 1;
		}
	}

	if (extrap != 0) /*need to extropolate data based on table of know values*/
	{
		fxubound = Tsat[jmax - 1];
		yubound = kL[0][jmax - 1];
		fxlbound = Tsat[jmax - 2];
		ylbound = kL[0][jmax - 2];
	}

	*ykL = ylbound + (yubound - ylbound) / (fxubound - fxlbound)*(fTsat - fxlbound);
}

/* Saturated Dynamic Viscosity
	Based on data tables recorded from NIST
	1 Input: Temperature K
	2 Outputs: Saturated Liquid Viscosity Pa-s, Saturated Vapor Viscosity Pa-s */
void mufTsat(double fTsat, double *ymuL, double *ymuV)
{
	double fxubound, yubound, fxlbound, ylbound;
	int j, extrap = 0;

	if (fTsat >= Tcrit) Error("Calculted bulk temperature above critical temperature in Function kfTsat");

	/*For saturated vapor-Tsat[j] corresponds to P[j]*/
	for (j = 1;j < jmax;j++)
	{
		if (Tsat[j] > fTsat)
		{
			fxubound = Tsat[j];
			yubound = muV[j][j];
			fxlbound = Tsat[j - 1];
			ylbound = muV[j - 1][j - 1];
			extrap = 0;
			break;
		}
		else
		{
			extrap = 1;
		}
	}

	if (extrap != 0) /*need to extropolate data based on table of know values*/
	{
		fxubound = Tsat[jmax - 1];
		yubound = muV[jmax - 1][jmax - 1];
		fxlbound = Tsat[jmax - 2];
		ylbound = muV[jmax - 2][jmax - 2];
	}

	*ymuV = ylbound + (yubound - ylbound) / (fxubound - fxlbound)*(fTsat - fxlbound);

	/*For saturated liquid-Tsat[j] corresponds to P[j]*/
	for (j = 1;j < jmax;j++)
	{
		if (Tsat[j] > fTsat)
		{
			fxubound = Tsat[j];
			yubound = muL[0][j];
			fxlbound = Tsat[j - 1];
			ylbound = muL[0][j - 1];
			extrap = 0;
			break;
		}
		else
		{
			extrap = 1;
		}
	}

	if (extrap != 0) /*need to extropolate data based on table of know values*/
	{
		fxubound = Tsat[jmax - 1];
		yubound = muL[0][jmax - 1];
		fxlbound = Tsat[jmax - 2];
		ylbound = muL[0][jmax - 2];
	}

	*ymuL = ylbound + (yubound - ylbound) / (fxubound - fxlbound)*(fTsat - fxlbound);
}

/* Compressed Liquid Properties
	Based on data tables recorded from NIST
	2 Inputs: Temperature K, Property Index (4 or 5)
	1 of 2 Outputs:
		(4) Thermal Conductivity W/m-K
		(5) Dynamic Viscosity Pa-s */
double compressed(double dfT, int retval)
```

```
{
	/*-Compressed liquid values are found using saturated liquid values at known T
	-A compressed liquid property value(rho(1), e(2), Cv(3), Cp(4), k(5), mu(6)) is returned depending on
what integer is
	 stored and passed for variable retval*/

	double yL, yV;

	/*function prototypes*/
	void kfTsat(double, double*, double*);
	void mufTsat(double, double*, double*);

	switch (retval)
	{
	case 4:
		kfTsat(dfT, &yL, &yV);
		break;
	case 5:
		mufTsat(dfT, &yL, &yV);
		break;
	default:
		Error("Invalid integer for retval in function compressed\n");
	}
	return (yL);
}

/* Superheated Vapor Properties
	Based on data tables recorded from NIST
	3 Inputs: Temperature K, Pressure Pa, Property Index (4 or 5)
	1 of 2 Outputs:
		(4) Thermal Conductivity W/m-K
		(5) Dynamic Viscosity Pa-s */
double superheat(double fT, double fP, int retval)
{
	double ll, lr, tl, tr;
	double Tubound, Pubound, Tlbound, Plbound;
	double yubound, ylbound;
	double jVl, jVr, yV;
	int i, ii, j, jj;

	/*Instead of extrapolating data, send Error if T>Tempmax or P>Pmax*/
	if (fT >= Tempmax) Error("Temperature exceeds maximum temperature table data in function
superheat");
	if (fP >= Pmax) Error("Pressure exceeds maximum pressure table data in function superheat");

	/*Bracket the upper and lower bounds for P*/
	for (j = 1;j < jmax;j++)
	{
		if (Pres[j] > fP)
		{
			Pubound = Pres[j];
			Plbound = Pres[j - 1];
			jj = j;
			break;
		}
	}
	for (i = 1;i < imax;i++)
	{
		if (Tem[i] > fT)
		{
			Tubound = Tem[i];
			Tlbound = Tem[i - 1];
			ii = i;
			if (ii == jj)
			{
				yubound = Tsat[jj];
				ylbound = Tsat[jj - 1];
				Tlbound = ylbound + (yubound - ylbound) / (Pubound - Plbound)*(fP - Plbound);
			}
			break;
		}
	}
	switch (retval)
	{
	case 4:
		ll = kV[ii - 1][jj - 1];
		lr = kV[ii][jj - 1];
		tl = kV[ii - 1][jj];
		tr = kV[ii][jj];
		break;
	case 5:
		ll = muV[ii - 1][jj - 1];
		lr = muV[ii][jj - 1];
		tl = muV[ii - 1][jj];
		tr = muV[ii][jj];
		break;
	default:
		Error("Invalid integer for retval in function superheat");
	}
	/*compute function in j direction*/
	jVl = ll + (tl - ll) / (Pubound - Plbound)*(fP - Plbound);
	jVr = lr + (tr - lr) / (Pubound - Plbound)*(fP - Plbound);
	/*using j values compute function in i direction*/
	yV = jVl + (jVr - jVl) / (Tubound - Tlbound)*(fT - Tlbound);

	return (yV);
}

/* Constants */
double A[32] = {
	1.150470519352900E1,
	1.055427998826072E3,
	-1.270685949968568E4,
	7.287844527295619E4,
	-7.448780703363973E5,
	2.328994151810363E-1,
	-1.635308393739296E1,
	3.730678064960389E3,
	6.299667723184813E5,
	1.210920358305697E-3,
	1.753651095884817,
	-1.367022988058101E2,
	-6.869936641299885E-3,
	3.644494201750974E-2,
	-2.559784772600182,
	-4.038855202905836E-4,
```

```
        1.485396303520942E-6,
        4.243613981060742E-4,
        -2.307910113586888E-6,
        -6.082192173879582E5,
        -1.961080967486886E6,
        -5.786932854076408E2,
        2.799129504191752E4,
        -2.381566558300913E-1,
        8.918796032452872E-1,
        -6.985739539036644E-5,
        -7.339554179182899E-3,
        -5.597033440289980E-9,
        8.842130160884514E-8,
        -2.655507264539047E-12,
        -4.544474518140164E-12,
        9.818775257001922E-11
};
double B[20] = {
        +0.916617720187E2,
        -0.179492524446,
        +0.454671158395E1,
        -0.658499589788E2,
        +0.734466804535E3,
        -0.682501045175E3,
        +0.631783674710E3,
        -0.539408873282E3,
        +0.430923811783E3,
        -0.300295738811E3,
        +0.156567165346E3,
        -0.504103608225E2,
        +0.720706926514E1,
        -0.123944440318E3,
        +0.140334800142E1,
        -0.211023804313E2,
        +0.173254622817E3,
        -0.444294580871E3,
        +0.138699365355E3,
        -0.235774161015E2
};
double D[7] = {
        4.8645813003E1,
        -3.4779278180E1,
        4.0776538192E2,
        -1.1719787304E3,
        1.6213924400E3,
        -1.1531096683E3,
        3.3825492039E2
};
double F[4] = {
        3.05300134164,
        2.80810925813,
        -6.55461216567E-1,
        1.59514439374
};
double G[17] = {
        6.1934792E3,
        2.9490437E2,
        -1.5401979E3,
        -4.9176101E3,
        6.8957165E4,
        -2.2282185E5,
        3.7990059E5,
        -3.7094216E5,
        2.1326792E5,
        -7.1519411E4,
        1.2971743E4,
        -9.8533014E2,
        1.0434776E4,
        -3.9144179E2,
        5.8277696E2,
        6.5409163E2,
        -1.8728847E2
};
double Gamma = 1.008854772E-3;
double RGC = 4124.299539;
double Rc = 31.36;
double Rtl = 77.0377;
double Rtv = 0.127454;
double Tc = 32.938;
double Tt = 13.8;
double Pt = 7042.09;
double T0 = 13.8, T1 = 35.0, T2 = 400.0;

/* Pressure
   Based on Equation of state (P-4) from Reynolds 1979
        2 Inputs: Temperature K, Density kg/m3
        1 Output: Pressure Pa */
double PfTR(double Tcell, double Rcell)
{
        double T[32];
        T[0] = Tcell;
        T[1] = sqrt(Tcell);
        T[2] = 1.0;
        T[3] = 1.0 / Tcell;
        T[4] = 1.0 / pow(Tcell, 2);
        T[5] = Tcell;
        T[6] = 1.0;
        T[7] = T[3];
        T[8] = T[4];
        T[9] = Tcell;
        T[10] = 1.0;
        T[11] = T[3];
        T[12] = 1.0;
        T[13] = T[3];
        T[14] = T[4];
        T[15] = T[3];
        T[16] = T[3];
        T[17] = T[4];
        T[18] = T[4];
        T[19] = T[4];
        T[20] = 1.0 / pow(Tcell, 3);
        T[21] = T[4];
        T[22] = 1.0 / pow(Tcell, 4);
        T[23] = T[4];
        T[24] = T[20];
        T[25] = T[4];
```

```
	T[26] = T[22];
	T[27] = T[4];
	T[28] = T[20];
	T[29] = T[4];
	T[30] = T[20];
	T[31] = T[22];

	double C[32];
	for (i = 0;i < 32;i++) C[i] = A[i] * T[i];

	double H[32];
	H[0] = pow(Rcell, 2);
	H[1] = H[0];
	H[2] = H[0];
	H[3] = H[0];
	H[4] = H[0];
	H[5] = pow(Rcell, 3);
	H[6] = H[5];
	H[7] = H[5];
	H[8] = H[5];
	H[9] = pow(Rcell, 4);
	H[10] = H[9];
	H[11] = H[9];
	H[12] = pow(Rcell, 5);
	H[13] = pow(Rcell, 6);
	H[14] = H[13];
	H[15] = pow(Rcell, 7);
	H[16] = pow(Rcell, 8);
	H[17] = H[16];
	H[18] = pow(Rcell, 9);
	H[19] = pow(Rcell, 3)*exp(-Gamma * pow(Rcell, 2));
	H[20] = H[19];
	H[21] = pow(Rcell, 5)*exp(-Gamma * pow(Rcell, 2));
	H[22] = H[21];
	H[23] = pow(Rcell, 7)*exp(-Gamma * pow(Rcell, 2));
	H[24] = H[23];
	H[25] = pow(Rcell, 9)*exp(-Gamma * pow(Rcell, 2));
	H[26] = H[25];
	H[27] = pow(Rcell, 11)*exp(-Gamma * pow(Rcell, 2));
	H[28] = H[27];
	H[29] = pow(Rcell, 13)*exp(-Gamma * pow(Rcell, 2));
	H[30] = H[29];
	H[31] = H[29];

	double P = Rcell * RGC * Tcell;
	for (i = 0;i < 32;i++) P += C[i] * H[i];

	return P;
}

/* Internal Energy
   Integration based on Equation (15) from Reynolds 1979
	2 Inputs: Temperature K, Density kg/m3
	1 Output: Internal Energy J/kg */
double EfTR(double Tcell, double Rcell)
{
	double U0 = 3.9275114E5;
	double Int_Cv0, Int_Cv01, SumCTdCI;

	double T[32];
	T[0] = Tcell;
	T[1] = sqrt(Tcell);
	T[2] = 1.0;
	T[3] = 1.0 / Tcell;
	T[4] = 1.0 / pow(Tcell, 2);
	T[5] = Tcell;
	T[6] = 1.0;
	T[7] = T[3];
	T[8] = T[4];
	T[9] = Tcell;
	T[10] = 1.0;
	T[11] = T[3];
	T[12] = 1.0;
	T[13] = T[3];
	T[14] = T[4];
	T[15] = T[3];
	T[16] = T[3];
	T[17] = T[4];
	T[18] = T[4];
	T[19] = T[4];
	T[20] = 1.0 / pow(Tcell, 3);
	T[21] = T[4];
	T[22] = 1.0 / pow(Tcell, 4);
	T[23] = T[4];
	T[24] = T[20];
	T[25] = T[4];
	T[26] = T[22];
	T[27] = T[4];
	T[28] = T[20];
	T[29] = T[4];
	T[30] = T[20];
	T[31] = T[22];

	double C[32];
	for (i = 0;i < 32;i++) C[i] = A[i] * T[i];

	double dT[32];
	dT[0] = 1.0;
	dT[1] = 1.0 / (2.0*sqrt(Tcell));
	dT[2] = 0.0;
	dT[3] = -1.0 / pow(Tcell, 2);
	dT[4] = -2.0 / pow(Tcell, 3);
	dT[5] = 1.0;
	dT[6] = 0.0;
	dT[7] = dT[3];
	dT[8] = dT[4];
	dT[9] = 1.0;
	dT[10] = 0.0;
	dT[11] = dT[3];
	dT[12] = 0.0;
	dT[13] = dT[3];
	dT[14] = dT[4];
	dT[15] = dT[3];
	dT[16] = dT[3];
	dT[17] = dT[4];
	dT[18] = dT[4];
```

```
	dT[19] = dT[4];
	dT[20] = -3.0 / pow(Tcell, 4);
	dT[21] = dT[4];
	dT[22] = -4.0 / pow(Tcell, 5);
	dT[23] = dT[4];
	dT[24] = dT[20];
	dT[25] = dT[4];
	dT[26] = dT[22];
	dT[27] = dT[4];
	dT[28] = dT[20];
	dT[29] = dT[4];
	dT[30] = dT[20];
	dT[31] = dT[22];

	double dC[32];
	for (i = 0;i < 32;i++) dC[i] = A[i] * dT[i];

	double I[32];
	I[0] = Rcell;
	I[1] = I[0];
	I[2] = I[0];
	I[3] = I[0];
	I[4] = I[0];
	I[5] = pow(Rcell, 2) / 2.0;
	I[6] = I[5];
	I[7] = I[5];
	I[8] = I[5];
	I[9] = pow(Rcell, 3) / 3.0;
	I[10] = I[9];
	I[11] = I[9];
	I[12] = pow(Rcell, 4) / 4.0;
	I[13] = pow(Rcell, 5) / 5.0;
	I[14] = I[13];
	I[15] = pow(Rcell, 6) / 6.0;
	I[16] = pow(Rcell, 7) / 7.0;
	I[17] = I[16];
	I[18] = pow(Rcell, 8) / 8.0;
	I[19] = (1.0 / (2.0*Gamma)*(1 - exp(-Gamma * pow(Rcell, 2))));
	I[20] = I[19];
	I[21] = (-pow(Rcell, 2) / (2.0*Gamma)*exp(-Gamma * pow(Rcell, 2)) + 1.0 / Gamma * I[19]);
	I[22] = I[21];
	I[23] = (-pow(Rcell, 4) / (2.0*Gamma)*exp(-Gamma * pow(Rcell, 2)) + 2.0 / Gamma * I[21]);
	I[24] = I[23];
	I[25] = (-pow(Rcell, 6) / (2.0*Gamma)*exp(-Gamma * pow(Rcell, 2)) + 3.0 / Gamma * I[23]);
	I[26] = I[25];
	I[27] = (-pow(Rcell, 8) / (2.0*Gamma)*exp(-Gamma * pow(Rcell, 2)) + 4.0 / Gamma * I[25]);
	I[28] = I[27];
	I[29] = (-pow(Rcell, 10) / (2.0*Gamma)*exp(-Gamma * pow(Rcell, 2)) + 5.0 / Gamma * I[27]);
	I[30] = I[29];
	I[31] = I[29];

	if (Tcell < T0) Message0("ERROR: Cell temperature is lower than minimum temperature %g\n",
Tcell);
	if (Tcell < T1) Int_Cv0 = G[0] * (Tcell - T0);
	if (T1 <= Tcell && Tcell <= T2)
	{
		double X = log(Tcell / T1);
		Int_Cv01 = Tcell - T1;
		Int_Cv0 = G[0] * (Tcell - T0);
		for (i = 1;i < 12;i++)
		{
			Int_Cv01 = Tcell * pow(X, i) - i * Int_Cv01;
			Int_Cv0 += G[i] * Int_Cv01;
		}
	}
	if (Tcell > T2)
	{
		double X = log(Tcell / T2);
		U0 += 3933318.541021109;
		Int_Cv01 = Tcell - T2;
		Int_Cv0 = G[12] * (Tcell - T2);
		for (i = 13; i < 17; i++)
		{
			Int_Cv01 = Tcell * pow(X, i - 12) - (i - 12)*Int_Cv01;
			Int_Cv0 += G[i] * Int_Cv01;
		}
	}

	SumCTdCI = 0.0;
	for (i = 0;i < 32;i++) SumCTdCI += (C[i] - Tcell * dC[i])*I[i];
	double Ecell = Int_Cv0 + SumCTdCI + U0;

	return Ecell;
}

/* Specific Heat
   Based on Equation (C-5) from Reynolds 1979
	1 Input: Temperature K
	1 Output: Specific Heat J/kg-K */
double CvfT(double Tcell)
{
	double Cvcell = 0.0;

	if (Tcell < T0) Message0("ERROR: Cell temperature is lower than minimum temperature %g\n",
Tcell);
	if (Tcell < T1) Cvcell = G[0];
	if (T1 <= Tcell && Tcell <= T2)
	{
		double X = log(Tcell / T1);
		for (i = 0;i < 12;i++) Cvcell += G[i] * pow(X, i);
	}
	if (Tcell > T2)
	{
		double X = log(Tcell / T2);
		for (i = 12;i < 17;i++) Cvcell += G[i] * pow(X, i - 12);
	}

	return Cvcell;
}

/* Density
   Root-finding method using function PfTR
	3 Inputs: Temperature K, Initial guess for Density kg/m3, Pressure Pa
	1 Output: Density kg/m3 */
double RfTP(double Tcell, double Rcell, double Pcell)
{
```

```
        double PfTR(double, double);
        double dR = 1e-3, dPdR;

        double tol = 1e-6, resid = 1;
        double Rcell_old, Rcell_new;

        int iter = 0;
        Rcell_new = Rcell;
        while (resid > tol && iter < 2)
        {
                Rcell_old = Rcell_new;
                dPdR = (PfTR(Tcell, Rcell_old + dR) - PfTR(Tcell, Rcell_old - dR)) / (2 * dR);
                Rcell_new = Rcell_old - (PfTR(Tcell, Rcell_old) - Pcell) / dPdR;
                resid = fabs(Rcell_old - Rcell_new) / Rcell_old;
                iter++;
        }
        return Rcell_new;
}

/* Saturation Pressure
   Curve fit based on Equation (S-3) from Reynolds 1979
        1 Input: Saturation Temperature K
        2 Outputs: Saturation Pressure Pa, Derivative of Saturation Pressure with respect to Saturation
Temperature Pa/K */
void PsatfTsat(double Tcell, double *Pcell, double *dPdT)
{
        double alpha = 1.5814454428;
        double X = (1.0 - Tt / Tcell) / (1.0 - Tt / Tc);

        *Pcell = Pt * exp(F[0] * X + F[1] * pow(X, 2) + F[2] * pow(X, 3) + F[3] * X*pow(1 - X, alpha));
        *dPdT = *Pcell*(Tt / ((1.0 - Tt / Tc)*pow(Tcell, 2)))*(F[0] + 2 * F[1] * X + 3 * F[2] * pow(X, 2) +
F[3] * (pow(1.0 - X, alpha) - X * alpha*pow(1 - X, alpha - 1)));
}

/* Saturation Temperature
   Root-finding method using function PsatfTsat
        2 Inputs: Saturation Pressure Pa, Initial guess for Saturation Temperature K
        1 Output: Saturation Temperature K */
double TsatfPsat(double Pcell, double Tcell)
{
        void PsatfTsat(double, double*, double*);
        double Psat_test, dPdT;

        double tol = 1e-6, resid = 1;
        double Tsat_old, Tsat_new;

        int iter = 0;
        Tsat_new = Tcell;
        while (resid > tol && iter < 10)
        {
                Tsat_old = Tsat_new;
                PsatfTsat(Tsat_old, &Psat_test, &dPdT);
                Tsat_new = Tsat_old - (Psat_test - Pcell) / dPdT;
                resid = fabs(Tsat_old - Tsat_new) / Tsat_old;
                iter++;
        }
        if (resid > tol) Message0("Saturation Temperature did not converge!\n");
        return Tsat_new;
}

/* Saturated Vapor Density
   Curve fit based on Equations (4,5,6) from Younglove 1982
        1 Input: Saturation Temperature K
        1 Output: Saturated Vapor Density kg/m3 */
double RVfTsat(double Tcell)
{
        double X = (Tcell - Tc) / (Tt - Tc);

        double fT = B[0] * log(X);
        for (i = 1;i <= 3;i++) fT += B[i] * (1.0 - pow(X, (i - 4.0) / 3.0));
        for (i = 4;i <= 12;i++) fT += B[i] * (1.0 - pow(X, (i - 3.0) / 3.0));

        double Rcell = Rc + (Rtv - Rc)*exp(fT);
        return Rcell;
}

/* Saturated Liquid Density
        Curve fit based on Equations (6,7,8) from Younglove 1982
        1 Input: Saturation Temperature K
        1 Output: Saturated Liquid Density kg/m3 */
double RLfTsat(double Tcell)
{
        double X = (Tcell - Tc) / (Tt - Tc);

        double fT = B[13] * log(X);
        for (i = 14;i < 17;i++) fT += B[i] * (1.0 - pow(X, (i - 17.0) / 3.0));
        for (i = 17;i < 20;i++) fT += B[i] * (1.0 - pow(X, (i - 16.0) / 3.0));

        double Rcell = Rc + (Rtl - Rc)*exp(fT);
        return Rcell;
}

/* Saturated Fluid Densities
   Hub for functions RVfTsat and RLfTsat
        1 Input: Saturation Temperature K
        2 Outputs: Saturated Liquid Density kg/m3, Saturated Vapor Density kg/m3 */
void RfTsat(double Tcell, double *rholiq, double *rhovap)
{
        double RLfTsat(double);
        double RVfTsat(double);

        *rholiq = RLfTsat(Tcell);
        *rhovap = RVfTsat(Tcell);
}

/* Property Calculator
   Hub for property functions
        4 Inputs: Temperature K, Density kg/m3, Pressure Pa, Property Index (1 - 5)
        1 of 5 Outputs:
          (1) Density kg/m3
                (2) Internal Energy J/kg
                (3) Specific Heat J/kg-K
                (4) Thermal Conductivity W/m-K
                (5) Dynamic Viscosity Pa-s */
double prop(double Tcell, double Rcell, double Pcell, int retval)
{
```

```
        double output;

        double RfTP(double, double, double);
        double RLfTsat(double);
        double EfTR(double, double);
        double CvfT(double);
        double compressed(double, int);
        double superheat(double, double, int);

        switch (retval)
        {
        case 1: /* Density */
                if (Rcell > Rc) output = RLfTsat(Tcell);
                else output = RfTP(Tcell, Rcell, Pcell);
                break;
        case 2: /* Internal Energy */
                output = EfTR(Tcell, Rcell);
                break;
        case 3: /* Specific Heat */
                output = CvfT(Tcell);
                break;
        case 4: /* Thermal Conductivity */
                if (Rcell > Rc) output = compressed(Tcell, 4);
                else output = superheat(Tcell, Pcell, 4);
                break;
        case 5: /* Dynamic Viscosity */
                if (Rcell > Rc) output = compressed(Tcell, 5);
                else output = superheat(Tcell, Pcell, 5);
                break;
        default:
                Message0("Property index %d not found\n", retval);
                break;
        }

        return output;
}

/* Mixed Phase Properties
   Linear interpolation of saturated properties based on the Vapor Phase Fraction AKA EOF
        3 Inputs: Saturated Liquid Property, Saturated Vapor Property, Phase Fraction/EOF
        1 Output: Mixed Phase Property */
double meanof(double A0, double A1, double EOFc)
{
        double meanA;
        meanA = EOFc * A1 + (1 - EOFc)*A0;
        return meanA;
}
```