AN IMPROVED HYDRONIC LOOP SYSTEM SOLUTION ALGORITHM WITH A ZONE-COUPLED HORIZONTAL GROUND HEAT EXCHANGER MODEL FOR WHOLE BUILDING ENERGY SIMULATION

By

EDWIN SCOTT LEE

Bachelor of Science in Mechanical Engineering
Oklahoma State University
Stillwater, OK, USA
2006

Master of Science in Mechanical Engineering
Oklahoma State University
Stillwater, OK, USA
2008

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
May, 2013

AN IMPROVED HYDRONIC LOOP SYSTEM SOLUTION ALGORITHM WITH A ZONE-COUPLED HORIZONTAL GROUND HEAT EXCHANGER MODEL FOR WHOLE BUILDING ENERGY SIMULATION

ACKNOWLEDGMENTS

My advisor: Dr. Fisher has guided me through research and personal matters over the last eight years, continually reminding me that joy can be found almost anytime.

My committee: I have been lucky to find a supportive group, whose critiques were clear and whose time and patience was greatly appreciated.

My friends: I have seen many come and go; too many to list them all. Dr. Sankar, a key to my success, was essentially my mentor for several years. He taught me what it was like to work sacrificially, helping others, and in the process learn so much.

My family: I have been blessed to have an incredible family. My parents watched me grow up, take off to Stillwater, and never seem to tire of school. Throughout this, they did not question my decisions once, instead showing ultimate support that I can only hope to match with my own family. On so many occasions, my brother dropped his own work to help whenever I needed, often turning my doodles into art. The family I married into blessed me with another set of loving parents, a great brother, sister, grandparents, and so many others.

My wife and kids: I am so thankful to have a wife who is supportive, kind, and loving even when the work days and weeks get long. Throughout these years, she stood by my side, helping nudge me along when I was stuck. She has been a constant source of pure and kind love for my children and me. I'm so excited to spend forever together with her and my two incredibly happy and smart boys.

My God: I am eternally grateful for the opportunity to work and study in such a blessed, comfortable, situation.

Acknowledgements reflect the views of the author and are not endorsed by committee members or Oklahoma State University.

To Gibson and Daxton, ...

Name: Edwin Lee

Date of Degree: May, 2013

Title of Study: AN IMPROVED HYDRONIC LOOP SYSTEM SOLUTION ALGORITHM WITH A ZONE-COUPLED HORIZONTAL GROUND HEAT EXCHANGER MODEL FOR WHOLE BUILDING ENERGY SIMULATION

Major Field: Mechanical Engineering

Abstract: Whole building energy simulation is a simulation platform which includes any number of aspects of a building. These tools are generally capable of simulating zones, air systems, hydronic systems, electric generation systems, among others. This simulation environment can be leveraged when investigating the integration between simulation domains. This research effort consists of three main foci, each of which is related to the concept of integration of simulation systems, within the context of whole building energy simulation.

A new hydronic system solution algorithm is developed and implemented in EnergyPlus which integrates the component and system simulation models, providing a flexible and robust simulation.

The effects of transport delay in a piping system are investigated, with experimental data being taken at a horizontal borehole test site. Experimental validation implies that transport delay effects can be predicted using a blended set of model results. Bounding studies demonstrate that the effects on a hydronic loop are less sensitive on a particular transport delay model, and more sensitive to the overall loop topology and configuration.

A ground heat exchanger model that integrates the hydronic simulation model, a ground simulation domain, and zone heat balance calculations is developed. The ground heat exchanger model was validated against experimental data to a high degree of accuracy using a coarse grid, providing a low computational burden suitable for implementation in a whole building energy simulation shell.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Algorithms

NOMENCLATURE

A	Surface area
c	Concentration
C_p	Specific heat
D	Diffusion Coefficient
E	Rate of evapotranspiration from the surface
e'	Vapor pressure deficit of the air
g	Acceleration due to an external field (gravity)
G_r	Net radiation into the surface
G_s	Soil heat flux from the surface
h_{fg}	Latent heat of vaporization
h	Convection coefficient, enthalpy
k	Thermal Conductivity
L	System (pipe) Length
m	Mass
$\dot{m}$	Mass Flow Rate
N	Number or count of a material or property
Nu	Nusselt Number
p	Pressure
Pr	Prandtl Number
$\dot{q}$	Heat Transfer Rate
Q	Volumetric Flow Rate
q''	Heat flux
Q'	Line source heat intensity per unit length
R	Thermal Resistance, Radial dimension
r	Radius
Re	Reynolds Number
R'	Thermal Resistance
T	Temperature
t	Time
$\overline{T}_s$	Annual average surface temperature
U	Surface conductance
$\vec{v}$	Velocity
V	Volume

x	General Direction
z	For ground heat transfer: depth from Ground Surface; In Bernoulli's Equation: Elevation above a reference, positive opposite the field (gravitational) force

Subscripts/Superscripts

0	Initial condition
a	Property of the air
c	Radial centroid
D	Dimensionless based on pipe diameter
eff	Effective
f	Property of the fluid
i	inner, arbitrary index
in	*Into* a cell or region of interest, or a property of a surface interior
∞	Free Stream
j	Arbitrary index
m	Mass
o	Outer
out	*Out of* a cell or region of interest, or a property of a surface exterior
p	Pipe
res	Residence Time
$region$	Property of an inter-partition mesh region
s	Property of the soil
$surf$	Surface
seg	Segment

Greek Symbols

α	Thermal diffusivity
γ	Psychrometric constant
δ	Slope of the saturation vapour pressure temperature relationship
$\overline{\Delta T_s}$	Average amplitude of surface temperature
$\Delta x, \Delta y$	Cell or domain length dimension
$\delta_{i,j}$	Kronecker Delta Function
ζ	Geometric series coefficient
θ	Phase Shift for minimum ground surface temperature
γ	Integration Variable
ρ	Density
τ	Coefficient to normalize units of time
λ	Second Coefficient of Viscosity
μ	Dynamic Viscosity
ν	Kinematic Viscosity
ρ	Density

CHAPTER 1

Introduction

Whole building energy simulation is used in many facets of industry, and found in a wide variety of research fields. In industry, these simulation tools are used in design optimization for minimizing cost, enforcement of standards and regulations, and evaluation of possible building and system configurations. In research, these tools are used to evaluate novel configurations and perform studies where the parameters may be intertwined in various components within the entire simulation shell.

Typical whole building energy simulation programs include the ability to simulate building zones, air systems, hydronic loops, electric power generation, among other things which will vary between applications. The level of interdependence between components varies, and the ability to model the interaction between components is dependent on the assumptions used in developing the models and the design/structure of the whole building energy simulation program shell.

1.1 Project Context: A Highly Integrated Foundation Heat Exchanger Model

The primary target of the current research is development of a generalized and highly integrated model for ground heat exchanger applications where the heat exchanger is in close proximity to a zone. When the heat exchanger is laid in the excavation area around the basement of a building, the configuration is referred to as a foundation heat exchanger. Full details on the development of the foundation heat exchanger model itself are available in Chapter 4.

For standalone, or loosely coupled, foundation heat exchanger models, there are limitations, which may include:

- A lack of feedback from zone heat balance calculations to the ground domain, where the zone heat balance provides a boundary condition for the ground through a heat pump or other system.
- A lack of feedback from the ground domain to zone heat balance calculations, where the ground temperature provides a boundary condition for the zone heat balance through a building surface.
- A lack of feedback to the central plant solution, where the fluid temperature response in the heat exchanger provides a boundary condition to other equipment on the hydronic loop.

Different models have been proposed, and are included in Chapter 4, however none of them include a full integration between the three domains:

- Zone heat balance
- Ground domain
- Central plant/Hydronic loop system

To ensure this level of integration can be captured in adequate fashion, the simulation shell must include certain prerequisites:

- A detailed zone heat balance
- A robust central plant simulation model
- A flexible connection between zone, air, and central plant components to allow diverse heat exchanger applications to be handled accurately. This includes the possibility of niche applications such as:
  - One-tube per trench heat exchangers (run-around systems)
  - Long heat exchanger runs, such as district heating applications, where the delay in the system is considered

The whole building energy simulation environment EnergyPlus is leveraged and used as the shell for developing the new ground heat exchanger model. EnergyPlus includes a detailed zone heat balance engine which has been validated in many cases, including experimentally. EnergyPlus also includes an existing central plant simulation model, however this model is lacking in robustness and flexibility. To ensure the entire simulation shell is capable of providing a suitable environment for development of the highly integrated ground heat exchanger model, a new central plant model was developed and a transport delay study was performed with experimental measurements to support modeling efforts. These comprise the three main aspects of the current work.

1.2 Outline

The general context of this work is a hydronic system containing a horizontal ground heat exchanger placed in the proximity of a building zone. This provides a focus on the integration of ground heat transfer, hydronic system simulation, and the zone heat balance. Within this context, the work is divided into three sections, each with an individual chapter:

Chapter 2: Flexible Simulation of Controlled Pumping and Piping Systems for Whole Building Energy Simulation: A new hydronic loop solver was developed and implemented as the EnergyPlus central plant system simulation model. A key feature of this work is a demonstration of abstraction from physical systems. The new modeling approach is referred to in this document as the Improved Continuity & Energy-Balance (ICE-B) model to distinguish from the existing EnergyPlus loop solver algorithms.

Chapter 3: Evaluating Fluid Transport Delay in Central Plants for Whole Building Energy Simulation: Transport delay in a horizontal ground heat exchanger was experimentally measured and used to evaluate approaches to

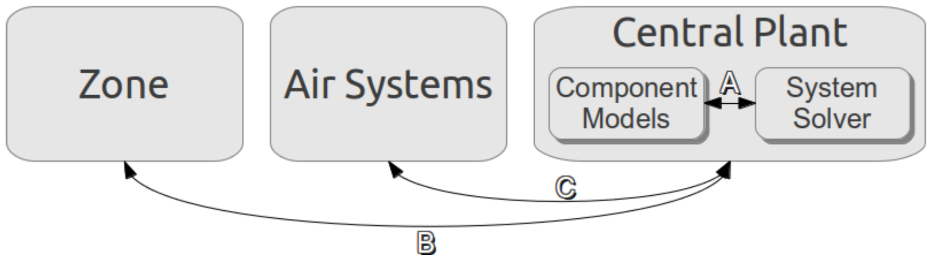


Figure 1.1: Dissertation topics and representation of integration points

model transport delay in the EnergyPlus central plant simulation model. Model development was performed using this data as a validation source.

Chapter 4: Efficient Horizontal Ground Heat Exchanger Simulation with Zone Heat Balance Integration: A new experimentally validated and computationally efficient horizontal ground heat exchanger model was developed and integrated with zone heat balance calculations.

The interaction between these three aspects is portrayed in Figure 1.1. The zone, system, and plant are already coupled within the whole building energy simulation environment, through the handling of zone loads and air system conditions. A major focus of this research (Chapter 2) involves improving integration between the internal component and system solver models of the central plant simulation, which is symbolized as integration **A** on the figure. While the ground is often involved in both the central plant simulation and the zone surface heat transfer rate calculations (floor, basement), the ground model itself is typically not coupled between these domains. The current work brings integration of the zone and central plant via the ground heat transfer model, symbolized as **B**. In addition, the transport delay study investigates an enhanced integration between the central plant and the air systems by capturing additional transients in the simulation, symbolized as **C**.

1.3 Research Summary

Each of the three main topics provide different research, with both experimental measurements and modeling efforts. While the previous section introduced the three topics individually, this section introduces the research in terms of the type of research performed.

1.3.1 Experimental Work

An investigation of the effects of fluid transport delay manifestations in the context of whole building simulation is covered in Chapter 3. As part of the research, experimental data was measured to provide a data set useful for guidance, development and validation for transport delay modeling. The experimental data uses horizontal borehole ground heat exchangers undergoing thermal response testing. The main feature of the data set is a high time resolution, with temperature, flow rate, and heat rate being sampled every second during the initial period of testing. This resolution allows the transport delay modeling work to be evaluated with a higher degree of certainty, as the transport phenomena timing is captured accurately.

1.3.2 Modeling Efforts

Each of the major topics in this research includes a modeling aspect. For the central plant simulation model, modeling work includes:

- A new solution algorithm to solve the system of state points, properly simulate components, provide a resultant flow distribution (operating point) in the system, and respect the interaction between pump model operation and the rest of the flow system.
- A redesign of the interface between component models and the new loop solution algorithm.

For transport delay, the modeling work includes:

- Development of a simulation testbed, to allow investigation of transport modeling work in an isolated environment, and to evaluate the suitability of experimental measurements.

- Implementation of an alternative transport delay model in the whole building energy simulation tool EnergyPlus, along with an evaluation of the significance of the delay model.

For the ground heat exchanger model, the modeling work includes:

- A generalized grid creation algorithm that allows for pipes and other domain features to be placed flexibly inside a Cartesian domain, and focuses computation in areas of high activity.
- Interaction between simulation domains, including the ground, zone heat balance calculations, and a hydronic system simulation model, including the effects of different time-scales.
- Interaction between coordinate systems, with a radial grid system in the near-pipe region.
- Validation of the model against numerous experimental data sets, as well as against an analytical solution in a simplified configuration.
- Implementation and validation of a two-pipe u-bend borehole heat transfer model inside the base model shell.

1.3.3 Unique Contributions

The unique features of this work include:

- A continuity & energy balance-based piping and pumping system simulation model that allows superb flexibility in terms of component placement, while still providing a robust, reliable model to be used in practice in system evaluation.
- Experimental measurement of transport delay in a horizontal borehole ground heat exchanger system.
- A generalized buried pipe heat transfer model applicable in a diverse set of applications.

- A computationally efficient approach to ground heat exchanger simulation.
- Integrated ground heat transfer model, zone heat balance, and hydronic. system simulation models

CHAPTER 2

Flexible Simulation of Controlled Pumping and Piping Systems for Whole Building Energy Simulation

Abstract

Whole building energy simulation tools are used to predict energy related features of all aspects of a building. In many cases, this includes a fluid loop, be it air, water or otherwise. Characteristics of these fluid systems are often investigated to optimize a design for minimizing energy use or provide insight into the interactions with other systems. An accurate simulation of pumping and piping fluid systems traditionally requires solution of a detailed pressure network, often utilizing a form of Bernoulli's equation. In addition to pressure-flow effects, other simulation elements include thermal activity, controls, a diverse set of possible components and complex topologies or configurations.

While a detailed pressure network solution provides an accurate flow distribution, certain characteristics of whole building energy simulation make the full network solution unattractive. These include the required input base of a detailed pressure network solution, and solution divergence that can come from poor initialization and solver instability. A new simulation model has been developed which does not rely on a full network solution. In place of the pressure network for flow resolution in the system, a predictor-corrector approach is employed. This approach provides a more robust simulation and reduces the required input parameters over traditional flow network solution algorithms. The new simulation model is termed the Improved Continuity & Energy-Balance (ICE-B) model, and is a replacement for the existing EnergyPlus fluid loop model. The ICE-B simulation model is demonstrated against a number of isolated test cases and "real-world" examples which show the model's ability to produce quality results and also the effort and process involved in abstracting a physical system using model paradigms.

2.1 Preliminary Discussion

The EnergyPlus central plant simulation model was first described by Fisher et al. (1999*b*). Several features from that paper still exist in the current model, however the simulation model has also evolved since the publication of that work. The new plant design proposed in the current work, termed the Improved Continuity & Energy-Balance (ICE-B) model, improves the capabilities of the overall plant simulation in terms of flexibility and robustness. The new plant simulation includes the following features:

- A novel plant loop solver (solution algorithm)
- Idealized control flow distribution algorithms
- Flexible interface between component models and the loop solver
- A robust overall system convergence algorithm

This chapter in laid out in the following fashion:

- This preliminary discussion presents the scope of the work. The physical system model is defined, followed by nomenclature and underlying assumptions.
- Discussion of different modeling approaches is included in a literature review. (Section 2.2)
- The simulation approach is described in a piece-by-piece fashion to build an understanding of the overall simulation model and contrast the existing and proposed models where applicable. (Section 2.3)
- Two "real-world" chiller plant systems are modeled, with an emphasis on the abstraction required to model these systems appropriately. (Section 2.4)

2.1.1 Staging

The methodology section of this chapter includes a detailed development of the plant simulation model. To enable an efficient discussion, the following sub-sections introduce different aspects of the simulation model, to stage the actual methodology discussion. The first sub-section (2.1.2) introduces the plant simulation in the context of a whole building energy simulation program. The second (2.1.3) describes the physical system that is to be modeled. The third (2.1.4) defines simulation state points, and how thermodynamic state points relate to the simulation model and solver algorithms. The fourth (2.1.5) is a special discussion focused on variable speed pumping. The final sub-section (2.1.6) provides an overview of the expectations of the model.

2.1.2 Whole Building Energy Simulation

Whole building energy simulation focuses on predicting the energy use of a building and its systems. These systems include those required to condition the building, but also include electrical generation measures among other things. Any of these systems may ultimately interact with a central plant simulation model. This "central plant" is a broad term that, for this work, represents fluid loops which are relevant to whole building simulation, including hot water loops, chiller plants, condenser loops, as well as heat recovery loops.

Three domains of a whole building energy simulation model which are utilized most frequently are the zone, air system, and central plant. The central plant model provides an ultimate link to the environment for many large systems through the use of, for example, air-cooled chillers, cooling towers, or ground heat exchangers. This link to the environment plays an increasing role as building energy is reduced, since it provides a constraint on the amount of heat transfer available for a given set of conditions. If the central plant is undersized and not able to reject the demand imposed on it, the result will be (for a chilled water system) an increase in the water temperature. As a result of increased chilled water temperature, the air system may not be able to meet the zone demand, impacting both thermal comfort and performance codes and standards.

An accurate central plant simulation model is a key part of an accurate whole building simulation model. However, as a part of a whole building energy simulation program, there are other facets to consider. The most prominent is the necessity for a robust simulation environment. The whole building energy simulation program and the underlying models are used by engineers and designers who may not have the tools to "debug" a problem with an underlying model[1]. At the very minimum, a simulation model must be able to provide realistic results that ensure mass and energy is conserved within the system, and provide meaningful assistance as procedural errors are encountered. The level of accuracy above this is dictated by the requirements and assumptions of the model.

Within a whole building energy simulation program, a plant simulation model may be used at varying levels of detail for generic simulation, building design, and research purposes. To accommodate this, simulation models often allow a varying level of input detail. This allows early-design work to include "big-picture" optimization and system evaluation, while also providing the means for performing detailed simulation with well-defined parameters such as in validation studies. In terms of plants, the early design work may include optimizing chilled and hot water loop configurations. Generalized flow network solvers require full inputs of pressure characteristics and control strategies to be employed for even simple simulations. The simulation model presented here balances flexibility with an understanding that minimizing the input burden maximizes the usability for many cases.

[1]This statement does not cover debugging problems related to erroneous user-input.

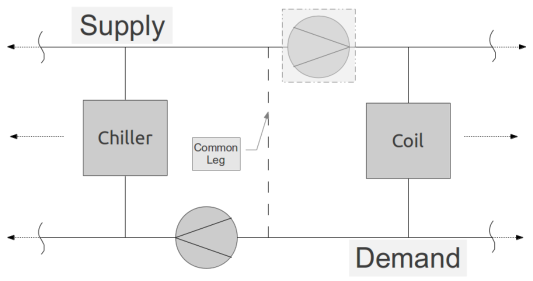
2.1.3 Loop Topology

To develop a system simulation model for a fluid loop such as a central plant loop, a clear definition of the physical and model topology must be established. Generalized solution algorithms may be capable of handling nearly any possible topology, however they have other limitations which detract from their usefulness. The new simulation model is based on the fixed set of topology rules that have been defined in EnergyPlus and used in the existing modeling approach, and were originally designed as described by Fisher et al. (1999*b*). These rules are flexible enough to handle many variations in plant systems, while including assumptions that ensure reliability from the simulation. This discussion provides the system topology definition, and is applicable to both the existing model and the new (ICE-B) model, unless otherwise specified.

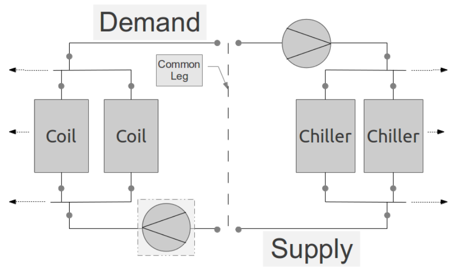
The EnergyPlus topology definition begins with a typical chilled water plant system, consisting of chilled water coils, chillers, and related pumping systems as shown in Figure 2.1a. The diagram shows primary pumping with optional secondary pumping accompanied by a common leg. This generic topology can be found in hot water systems also, as well as condensing systems. The use of coils and chillers in the diagram is intended to keep the discussion grounded to a certain level of physicality. The diagram shows two chillers and two coils as one possible configuration, however supplementary components are implied by the use of rays[2]. One facet that is missing from the diagram is the use of "non-pumping series" components, such as multiple chillers in series, or the addition of an economizer in series with the other components. These configurations *are* included in both the existing and proposed models, although the existing model had difficulty handling these cases due to improper communication between the system and component simulation models.

The system schematic is now portrayed differently without affecting the loop topology itself, transforming from Figure 2.1a to Figure 2.1b, with the pump symbols

[2]These rays, or arrows, are used on diagrams throughout this topology discussion as a representation that other components may be placed, but are not shown, on the current system.



(a) Simplified, common representation of a physical chilled water loop system



(b) Chilled water loop system in model visualization

Figure 2.1: Initial abstraction of a common chilled water loop system into model form

implying flow direction. The coils and remainder of a "demand side" are drawn as a parallel set of components with an inlet splitter preceded by an inlet leg and an outlet mixer followed by an outlet leg. The chillers which represent the "supply side" components makeup a similar path on the opposite side.

Component placement is more flexible in the ICE-B model than the existing model because of the solution algorithms in place in the new model. The diagram in Figure 2.1b shows pumps placed at each side of the loop, on the inlet leg, followed by a series of components in parallel. Many component types are available in the existing EnergyPlus component model library, which is one reason EnergyPlus was selected as a development platform. Although the pumps are placed on the inlet leg of the loop in Figure 2.1b, the ICE-B model allows pumps to be placed flexibly around the loop. In the existing model, pump placement is limited to very specific locations, which may increase the difficulty in abstracting a physical system into the simulation model form. Pump placement in the proposed model has one limitation:

> Each independent flow path may have at most one pump component.

This limitation is due to the pump model specifics, which are covered in section 2.3.3. In addition to pumps, the components which are shown in the parallel sections may also be placed in the entrance and exit legs surrounding each parallel set of components. One key application of general component placement is the use of an economizer, in which a heat exchanger may be installed in a variety of places in the loop to provide improved efficiency with a carefully selected economizer approach temperature. In the existing model, these economizers could be placed at essentially any location on the loop, however they could not be controlled properly because the loop solution algorithms did not communicate properly with the component model. In the ICE-B model, the economizer is provided proper boundary condition data to allow it to be utilized more reliably.

The symmetry of the system is clear as shown in Figure 2.1b. The discussion will now be narrowed to an individual "loop-side," either the demand *or* supply, with the common pipe set aside temporarily. By generalizing the loop-side to contain any type of components in parallel, the resulting loop-side is shown in Figure 2.2a. This loop-side consists of an inlet section, followed by a series of parallel sections, followed by an outlet section. In the existing model, each of these loop-sides used individual solution algorithms: a demand solver and a supply solver. In the ICE-B model, the solution algorithms are generalized to accommodate any single loop-side, improving code reuse and reducing the code maintenance burden. Figure 2.2a includes the following features:

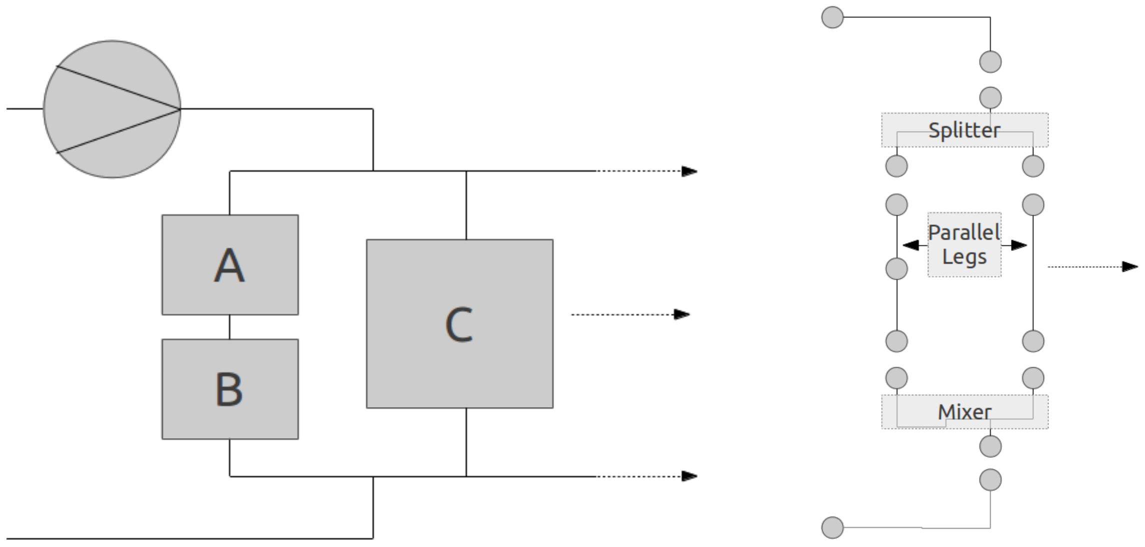
- The inlet leg contains a pump, which may be common for many systems, but is not a required part of the topology for the ICE-B model. Instead this leg may contain any number of other components, including no physical components at all[3]. In the existing model, there are much stricter restrictions on the placement of pumps and other components, especially on the inlet leg of each loop-side.
- The parallel section may contain any number of paths; two are explicitly shown on the diagram with the generalization being implied to be greater than two. In addition, the diagram does not show that the number of parallel paths can be one. This would result in all loop-side components being in series.
- Each parallel path may actually contain any number of series components.

[3]A connector object is required to act as a placeholder for flow segments without any components.

- The outlet is similar to the inlet leg and may contain any number of series components.

If we introduce the term *branch* to represent a collection of series components, then the topology of a single loop-side can be summarized succinctly:

1. An inlet branch
2. A collection of one or more parallel branches
3. An outlet branch



(a) Component form, with a pump and parallel components (b) Generic branch topology

Figure 2.2: A single loop-side in model arrangement

For simulation purposes, the loop-side shown in Figure 2.2a is now broken into individual branches as shown in Figure 2.2b. The series and parallel sections are connected using a flow splitter and a flow mixer. The assignment of a predefined flow splitter and flow mixer in the system is inline with a key assumption in place in this simulation model:

> The flow direction in each branch of the simulation model is predefined, stemming from the assumptions inherent in the abstraction of the original physical system into simulation model form.

The predefined flow direction is certainly a major assumption of the system, and this will be leveraged in developing the system solution algorithms. While components can be placed flexibly around the loop, having a predefined flow direction allows the solution algorithm to be robust and focus on variable flow distributions without needing to solve flow direction simultaneously.

The form shown in Figure 2.2b is now reoriented into Figure 2.3. The inlet and outlet state points are shown on the diagram (ψ_i), with the inlet being a boundary condition for the loop solution algorithms. In addition, the flow direction is added to the diagram to be clear that this is a predefined part of the system simulation model assumptions moving forward.

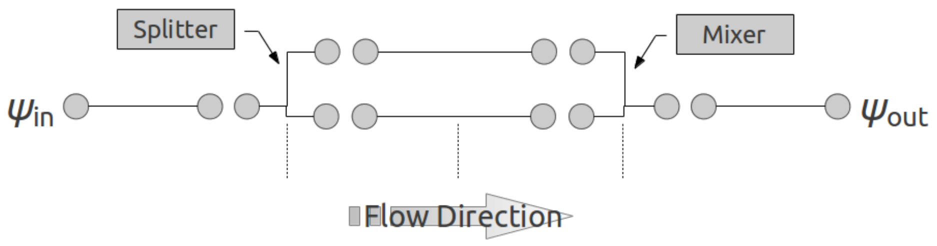


Figure 2.3: Final visualization of the loop-side to be solved by the simulation model

The inlet and outlet state points are shown in Figure 2.3 to provide a broad view of the system. However, each branch inlet and outlet, and even intermediate points on branches (for branches with multiple components), are all state points to be solved by the simulation.

2.1.4 State Point

A thermodynamic state point represents the state of a fluid at any point in space and time. A thermodynamic state includes fluid pressure, temperature and density, as well as perhaps other properties depending on the fluid or mixture. In the simulation, there is additional metadata which can be attached to a fluid state. A detailed description of the original state point design for EnergyPlus is found in Fisher et al. (1999*b*). Some additional features include:

- Fluid Mass Flow Rate: While this is not a property of the fluid, it is a property of the system at a given point. Simulations which do not capture transient flow phenomena do not require this to be defined at every intermediate point, but only once per independent flow path. However, there are development and maintenance reasons to keep this data stored at intermediate points, most notably the ability to debug individual component models and ensure continuity throughout the system.
- Historical Data: While the current fluid state is defined from a set of properties at each point, there are many component simulation models, along with the system solution algorithms themselves, that consider thermal history. While individual models could store this data, it is more useful to store it in a central location. While the existing simulation model had some historical data saved, the new model added a number of historical metadata to allow improved control of system convergence.
- Solver Metadata: The proposed ICE-B simulation model added several variables to the previous simulation state point structure. The most notable is the requested mass flow rate. This is used by a component to issue a request for flow to the solver algorithms, and also in convergence monitoring. (See section 2.3.2.2.)

Since the fluid half-loop consists of a collection of components connected with these state points, solving the system implies solving for the state point properties for the entire half-loop. This is a core feature of the solution algorithm development as described in following sections. Note that there is a distinction between thermodynamic state and flow properties vs. historical/metadata information, which need not be solved.

2.1.5 Variable Speed Pumping

The solution of thermal piping systems is nontrivial, even for simple systems. Certain component attributes make the solution much more difficult, including the simulation of variable speed pumping. Variable speed pumping systems are widely accepted in low energy plant applications and allow the system to reduce operating conditions under a part load state. While there are simulation tools capable of simulating variable speed systems, there are special difficulties encountered in providing them in a generalized and robust tool, including how the increased system dimension is handled, along with the increased input requirements. The existing EnergyPlus central plant simulation model included variable speed pumping, however there were significant limitations, including the ability of variable speed pumps to ramp down under all part-load conditions due to a lack of communication between the pump algorithms and the loop solution algorithms. The new ICE-B model uses enhanced request and flow distribution algorithms to ensure variable speed pumping can be simulated properly, even with primary-secondary or dedicated pumping systems.

2.1.5.1 System Dimension

A constant speed pump in a pressure network will operate at a fixed rotation rate resulting in a flow that varies according to the system head. A variable speed pumping system varies the rotation rate in order to minimize energy use under part load conditions. This speed must be controlled, typically by a pressure (or pressure differential) measurement.

Neither the existing, or the new ICE-B plant simulation model utilize a full pressure network simulation. Without a pressure network, pump models are much different from those used in generalized pressure solutions. Including variable speed pumping into the simulation environment introduces additional variables which must be solved, increasing the system dimension. This may also introduce unstable interactions between components as the pump as well as individual components attempt to resolve to a minimum flow condition while still meeting loop demands.

The pumping model implemented in the simulation utilizes a technique that allows variable speed pumps to “float” in the system. Instead of driving the flow, they simply respond to the demands of the loop. This is in contrast to constant speed pumps, which tend to provide flow regardless of demand.

2.1.5.2 Input Requirements

In a simulation, the addition of variable speed pumping requires additional or modified controls. For a full pressure network simulation tool based on a generalized equation solver such as Modelica (Modelica Association, 2010), a control profile must be added to the simulation to ensure it is stable throughout the simulation. The simplified flow network solution and the interface between components (including pumps) and the solution algorithms in the proposed work provides a simple mechanism for allowing variable speed pumping, with only a minimal increase in the input specification.

2.1.6 Scope and Purpose (Assumptions)

Solving the system of state points and underlying governing equations can be performed using a variety of approaches. The level of accuracy and detail provided by the solution algorithm is often tailored to a specific problem. In some pressure-network based piping system solvers, thermal conditions may be ignored to focus attention to the pressure distribution. This does not indicate that the fluid temperature in the system does not vary, it simply decouples that problem from the pressure distribution, and essentially implies a trivial solution to the fluid temperature problem. For the applications of the current model, this assumption is not valid. As the current simulation model is implemented within a whole building energy simulation program, one major desired outcome of this model is energy use. This emphasizes the solution of the flow and temperature distribution in the system. The pressure network is the fundamental driving force of a physical system like those covered by this simulation model. However, the simulation model is able to disconnect the pressure network from the flow and temperature distribution solution, and there are significant benefits:

- Reduced dependency on detailed pressure inputs
- Improved simulation robustness
- Implied underlying assumptions for prescribed flow direction (section 2.1.3)

Ultimately, the goal of this work is not to produce another pressure network solver, which has been done many times before. Instead, the goal of this work is to create a fluid loop simulation model that uses ideal control assumptions, continuity and energy balance techniques and rule-based flow distribution to provide a robust environment that does not rely on detailed pressure-network information to solve the governing equations. The model is developed based on the loop topology rules already in place in EnergyPlus, but introduces enhanced flexibility in how components are placed and controlled compared to the existing model. Compared to the existing model, the new model also has improved convergence monitoring and interaction with other simulation domains such as the air-system components and system simulation models.

2.2 Literature Review

There are many methods available for representing a physical pumping and piping system or central plant as a simulation model. An initial review is provided for generic pumping and piping system solution techniques (2.2.1) to lay the baseline for sections following related to thermally active systems such as central plants. The solution of these thermally active systems is classified in two main categories: lumped energy balance representation (2.2.2) and algorithmic solution techniques (2.2.3).

2.2.1 Pressure-based Flow Networks

The most abstract flow model is a numerical solver that does not directly provide a physical modeling layer. An example is the Modelica (Modelica Association, 2010) language format which has been implemented in a number of programs, in both private industry and open-source forms. In these applications, the level of abstraction from physicality is generally low (physics are modeled in detail), requiring a higher level of detail in model code, inputs, and complexity. Libraries, including the open-source Buildings library from Lawrence Berkeley National Laboratory (Wetter, 2009), provide a portion of the overhead for particular languages/applications.

Pressure-based flow network models generally solve a form of Bernoulli's equation. For steady incompressible flow and negligible viscous friction, this equation is provided as:

$$\frac{\vec{v_1}^2}{2} + gz_1 + \frac{p_1}{\rho} = \frac{\vec{v_2}^2}{2} + gz_2 + \frac{p_2}{\rho} \tag{2.1}$$

By ignoring the elevation in the system, including the mechanical energy exchanges, and reformulating in terms of the pressure head, the equation is offered as:

$$\frac{p_1}{\gamma} + \frac{\vec{v_1}^2}{2g} + h_{pump} - h_{minor} - h_{friction} = \frac{p_2}{\gamma} + \frac{\vec{v_2}^2}{2g} \tag{2.2}$$

From this point, terms can be added or removed from this energy balance based upon further assumptions and additional system physics modeling. Whatever the formulation, mass and energy balances are enforced, in either steady or transient formulation, to result in a flow network distribution.

The use of a flow network has an extensive set of applications, across a wide variety of industries. As such, the literature is filled with simulation and case studies. Many solutions are based on the method designed by Cross (1936) or linear theory described by Wood and Charles (1972). Collins (1980) described these along with the Newton-

Raphson technique in the context of guiding engineers toward a suitable approach for solving piping network problems. Todini (2006) performed a detailed analysis of various numerical algorithms to solve piping networks, evaluating the iteration cost and dimensionality of each method. Oke (2007) used statistical analysis to evaluate the three major simulation techniques.

Gay and Middleton (1971) utilized graph theory and matrix transformations in creating a diakoptics solution to the network piping problem, which is claimed to produce a faster solution. Higson (1984) justified the use of linear theory to solve piping networks with nodal heads, while Haghighi et al. (1992) applied linear theory to a variety of network components to allow for more generalized solution capabilities.

Within an iterative piping network solution, Lang and Miller (1981) described the benefit of using a smooth friction factor correlation to speed up and stabilize convergence. HaktanIr and ArdIclIoglu (2004) created a pipe network solution algorithm using a numerical implementation of the Darcy-Weisbach friction factor expression. Preece and Ti (1989) utilized a fictitious branch during network solution to solve an equivalent network under mixed boundary specification conditions. Nielsen (1989) exercised various numerical formulations to show stability and convergence, showing that it is better to have a pipe-discharge based flow equation than head-based. Lopes (2004) performed a specialized development and implementation of the Hardy-Cross method. A novel approach was used by van Zyl et al. (2008) to approximate the pipe head loss relationship using a gradient method for solution.

Boulos and Wood (1990) used an explicit formulation of pipe network equations which allowed the system to contain a variety of constraints, such as required flow through various legs of the system, or a required head at a given node. Boulos and Altman (1993) applied explicit formulation to systems with closed network legs to allow simulation of varying topology during system operation.

Mohtar et al. (1991) created a simulation program using a finite element model of different pipe network components that allowed for generalized network solution. Kohlenberg and Wood (1994) simulated and validated a detailed power plant flow network simulation. Zhu et al. (2012) utilized a dense three dimensional finite element approach, directly solving the Navier-Stokes equations for the simulation of a pumping station.

Berghout and Kuczera (1994) used an iterative approach to network simulation using linear programming techniques, capable of simulating components that do not have a fixed pressure head-flow relationship.

Nazeer et al. (1999) used a detailed mathematical development to solve a flow network with a centrifuge, leveraging sparse matrix techniques in the solution algorithm. Leung et al. (2000) used flow network analysis along with safety and failure estimation to predict serviceability and maintainability of a system. Estrada et al. (2009) described advances to a network model, including specialized water irrigation components, which is solved with a matrix solution. Ie et al. (2001) developed a simulation architecture with two sub-layers: a hydraulic analysis to calculate the response of the flow system, and a controls sublayer to implement system controls. Hodge (2006) utilized proprietary software to solve piping network problems in the context of engineering education.

This review shows that the simulation of piping systems using a flow network is a diverse topic, with many different possible formulations. Much research has been done on this topic to make the models more applicable with improved reliability, flexibility, and computational efficiency. These are important characteristics for simulation models in whole building energy simulation. For thermally active systems, and especially controlled thermal systems, additional simulation layers are required.

2.2.2 Energy Balance Approach

The energy balance approach is based on a level of abstraction where the flow network need not exist in the model. Instead, the energy exchanges in the system are based solely on a movement of heat between components and loops. Initially, a load can be applied to a fluid loop as a demand. The thermal energy equipment on the loop may be as simple as a single chiller or heat exchanger, for example. If the equipment performance model is based solely on loading conditions (design and actual), independent of fluid approach conditions, then the part load ratio can be determined directly, without the need for a fluid simulation. Consider the following generalization as an example:

$$PLR_{e-b} = \frac{\lambda_{actual}}{\lambda_{design}} \tag{2.3}$$

$$\epsilon = f\left(PLR_{e-b}\right) \tag{2.4}$$

where:

PLR_{e-b} :Generic part loading of a system using the energy-balance

λ :A system parameter, such as heat transfer rate

ϵ :System efficiency, used for energy consumption calculations

The equipment energy usage is then calculated by this part load ratio and performance data. If the equipment is connected to other loops, the response on the other loops can be calculated using the same approach. Pumping power can be estimated in a similar fashion.

The lack of a fluid loop eliminates the possibility of capturing temperature dependent phenomena on the loop, including approach temperature dependence of component efficiency. If the components are assumed to operate at design flow rate then there is no possibility for variable speed pumping energy savings. Controls are difficult to mimic in such cases because in a real plant where it will operate on a setpoint temperature, there may be cases where the plant may be in a dead-band condition. However, this cannot be simulated in a fluid-less simulation. There is also a lack of specialized feedback from the loop to the other components, for example the possibility of temperature dependent economizer operation is eliminated. While the design capacity may be used as a limiting situation for undersized plant systems, any fluid temperature limits cannot be addressed. This method lacks the ability to feed useful temperatures back to the coils or other demand equipment to calculate proper heat transfer rates for demand equipment.

While this method has limitations, it also allows for a relatively small input footprint. Design values, part load coefficients and power calculation coefficients (or parameters) can be used to get a first-order approximation of plant energy use. In addition, simple control strategies can still be used in order to perform load dispatch and component staging. This can allow a very high-level view of the effects of energy savings available in optimizing control strategies, and this simulation can be driven with a building load profile to create a suitable prediction.

The building simulation program BLAST (Witte et al., 1989; Taylor et al., 1991) included an integrated simulation approach for zone, air systems, and plant systems simulation, which utilized a form of this energy balance approach. The literature shows that a majority of other thermally active system simulations are also based on an energy balance approach, as described next. This is an expected conclusion, as these methods do not require the overhead of a flow simulation while still provide an approximation of the energy effects.

A simulation model by Braun (1992) used an energy balance approach to optimize chiller and ice-storage strategies by creating a simple representation of the chiller, ice storage model, fan and coil, cooling tower, and control strategies. The power for the entire cooling plant is a function of chiller load, ambient wet-bulb (for the cooling tower model), and the chiller supply temperature; along with a set of empirical coefficients. The chiller supply temperature is approximated based on ice storage mode, bulk thermal storage temperature (assumed constant), and effective heat capacitance. The models are driven by a set of building loads, and use a number of design conditions for the system including required flow rate. This provides an example of using a simplified approach to perform high-level optimization before rigorously modeling the detailed system. Braun (2007) provided an optimization of control strategies for hybrid chiller plants by creating simplified models of each component and then optimizing the chiller sequencing along with other attributes.

Numerous other optimization studies have apparently used a simple energy balance approach[4]. This is again expected as optimizations which take many trials benefit (in terms of computation burden) by using the simplest simulation approach. Chen et al. (2007) utilized a set of empirical regression based models to provide a representation of a central plant for creating a plant optimization program and optimizing a campus chiller plant. Hydeman et al. (2002) used the BLAST simulation program to create loads and created a standalone spreadsheet implementation of a plant model to evaluate fan speed control requirements for standards development. Nelson (1999) described a simulation study of a primary/secondary chilled water system addressing physical phenomena such as load degradation on time due to equipment age, however the system simulation model was not described in great detail. A simplified plant simulation was created by Sakamoto et al. (1999) and used within a genetic algorithm based optimization for district heating and cooling plant operation. Wang et al. (2007) used a chiller component model inside an optimization for chiller plant operating strategies. Sun (2010) optimized control strategies in a

[4]Not all sources were fully clear on modeling details, but the context implied that an energy balance approach was utilized.

multiple chiller plant simulation model.

King and Potter (1998) demonstrated an unusual approach to network simulation. A series of plant component models were developed for optimizing ice storage control strategies. A set of plant operation equations was presented. The equations to be solved at any given condition varied. A table in the reference lists which equations were solved for each system condition, and the variable being solved for in each equation. The equation system varied significantly within the same simulation model, based on the current operation mode. This reference demonstrated that even within an apparently energy balance based approach, different scenarios lead to a different system of equations and solution strategy. Essentially a template was set up for each condition.

2.2.3 Algorithmic & Templates

Energy balance styled central plant simulation models are often developed for a single or minimal set of possible configurations. This restriction allows for a simple, well-defined system of equations, suitable for use within an optimization and to perform first-order prediction of overall energy use. The DOE2 whole building energy simulation program demonstrated an approach where a building simulation program included a number of common configurations, and allowed the user to make minor changes to the components, but not affect loop topology or overall control strategies.

Hunn (1979) provided a comprehensive introduction to the design and capabilities of the initial release of the building simulation program DOE2. The program consists of four subprograms. The loads subprogram utilizes a zone temperature which is assumed to be a known value and is used in load calculations. The systems subprogram performs the simulation of building air systems, and also post-processes the zone temperature to approximate conditions where the system did not precisely meet the building load. The systems subprogram included 16 pre-programmed space-conditioning systems, allowing minimal input for a majority of simulation needs, but limiting the usability in novel applications. A plant subprogram performed the simulation of central plant equipment, while an economics subprogram processed the resulting energy use through economic calculations. Buhl et al. (1985) described a number of new features of the DOE2 simulation engine, including improved cogeneration handling in the central plant simulation. Bahel et al. (1989) validated the simulation program DOE2.1A against measured data and compared to the capabilities of a main-frame simulation program available at the time. Pasqualetto et al. (1998) also performed a number of tests with DOE2.1, including inter-model comparison and empirical validation. Sekhar and Yat (1998) used DOE2.1E to compare a number of air-conditioning systems in a large office building, including a chilled water plant. Tian et al. (2009) used DOE2.1E as a baseline in developing a model of a high performance building, then used this baseline to build a more advanced model in EnergyPlus (Crawley et al., 2001), including features that were not possible in the baseline.

For a majority of building simulation scenarios, the central plant design may be based on standardized configurations that have been proven successful in industry. While every configuration will differ by required equipment capacities and design flows, this template approach allows the user to quickly select a configuration, input a limited amount of data, and predict a reasonable response of the system. For model development, a benefit of this template approach is a well-defined causality within the system. With a preset topology, the embedded controls have a well-defined set of actuation points. Common features may be difficult to implement including variable speed pumping due to the highly predefined interactions between system components.

Template approaches are useful for a majority of standardized configurations. However, templates are not suitable for novel configurations or novel control strategies, as a new template must be created for each new configuration. Systems which are tightly coupled from the zone to the air system, through the central plant, and finally back to environmental (condenser) heat transfer equipment are also expected to have difficulty due to the generalized and coupled nature of the system. A more general approach must be in place to simulate novel and coupled systems.

2.2.4 Existing EnergyPlus Model

The existing simulation model which is used in EnergyPlus was described by Fisher et al. (1999*a*). Since then, many changes have been made as the program has evolved to accommodate new systems and applications, as well as an ever-increasing and diverse library of component models. Fisher et al. (1999*a*) described the system simulation model, how components are called by a manager interface, and how the solution methodology allows for a generalized component topology which then defines the type of system being simulated.

The core concepts of the original design such as the loop topology and disconnected loop-sides are being used in the new ICE-B model, although many advances are made in terms of component flexibility and simulation robustness.

2.2.5 Discussion

Template simulations provide usability for engineers and designers to simulate standardized systems. The input requirements for such systems are generally limited to design conditions, which further increase the usability. Since the systems are well-defined, testing can be performed to ensure that the solution is robust over a great range of conditions. In contrast, generic numerical solvers provide the ultimate in flexibility for experienced researchers investigating novel configurations and complex control systems. The input requirements will generally be higher because more physical phenomena is simulated. For flow network solutions, pressure characteristics are required for each independent flow path, at a minimum. For controls, the input requirements will differ greatly between control strategies.

An alternative approach utilizes predefined template flow paths and control configurations, however uses an underlying generalized solver to calculate state variables and predict the system response. This is an important use of such solvers, but relies on an intermediate layer, or application. This application could be intelligent and develop these templates, but expose only certain parameters of the system, limiting the possibility of numerical problems. In addition, the errors that may be encountered in the abstract numerical solver could be given context with this intermediate physical layer. The program would be able to interpret these errors and provide useful information for fixing the error. Obviously this would differ in every application, but consider an example as the difference between:

```
Error: Instability detected in block 3;
       Check constraints
```

and:

```
Error: Loop flow rate out of bounds;
       Check design flow for components A and B
```

The first form exemplifies the output of an abstract numeric solver. The second form may represent the output from a hybrid solver/template, where the error can be interpreted much better by users. This idea blurs the definition of a template system, as the key is whether or not the underlying solver is a template, or if the inputs are merely limited to template formulated combinations. The current work does not use predefined templates, instead allowing a high level of flexibility in terms of component placement/loop topology, and controls, with the topology only limited to the form shown in Figure 2.3. The current work does not utilize a generalized solver to solve the system of resulting equations, but instead relies on a predictor-corrector mechanism, in conjunction with a flow-wise solution algorithm to solve the system response and perform controls operations (see section 2.3).

2.2.5.1 Simulation-Based Control

In any solver strategy, robustness is a major factor, especially in whole building energy simulation applications. Simulation-based control is emerging as an active area of research not only for general controls, but also for real-time optimization in a diverse set of fields within industry. A study of using simulation-based control to operate an industrial injection molding process was described by Johnston et al. (2009). Lee and Prabhu (2010) described an optimized route-planning algorithm that relied on simulation to minimize energy costs related to delivery services. Within any simulation-based controller, the robustness of the simulation algorithm is of highest priority. There is no opportunity to debug problems once the algorithms are embedded and operational. Thus, the simulation algorithm must be stable and convergent. Certainly some abstract numerical model simulations are reliable, but the breadth of the simulation is so large, it is difficult to predict that in every case the simulation will be successful. Within a tighter, more well-defined simulation algorithm, these conditions are easier to detect and logic can be implemented to bring the simulation to a stable condition. The loop topology variations available in the proposed model are intended to cover a vast set of possible hydronic loops, both conventional and novel, while still providing mathematical systems that can be solved robustly under the given assumptions, allowing it to be suitable for simulation-based control applications. The implied flow direction in the loop is a key feature of this robustness.

2.3 Methodology

The proposed ICE-B simulation model described here is developed based on the topology shown in Figure 2.3 with the intention of solving the state point (temperature and flow) distribution of this network. This implies a solution of the governing continuity and energy equations. The momentum equation holds an interesting position in this solution due to the lack of a full pressure network and the predefined flow directions in the system. The solution of the momentum equation is inherent in the flow distribution algorithms, but not does appear explicitly in a typical pressure-flow relationship form.

Solving this system also implies solving for the energy usage of the system via the individual component models, which provide the link (energy equation) between state points. Controls are employed in the system which attempt to satisfy various setpoint temperatures around the network as needed. Note that in many cases, the "loop solver" is referred to, which comprises the solution algorithms for a single half-loop (Figure 2.3). This half-loop, or loop-side encompasses one side of a physical loop. (See section 2.1.3).

2.3.1 Component Model

The system of state points scattered through the network shown in Figure 2.3 (the small dots) are linked together by component models. The form of the component model can vary from empirical curve fitted chiller relationships to ideal pipes that pass fluid states directly from inlet to outlet. (These pipe models are used as placeholder components in the system, making connections in loop spaces where no physical components exist.)

Component models have two responsibilities:

1. Solving the continuity equation
2. Solving the energy equation

While this may seem like a curious emphasis to make, it is crucial to an understanding of the design of a robust simulation model. The boundary conditions for each equation may be fixed for a component model, or vary through the simulation. The component model is responsible for enforcing both a mass (continuity) balance and an energy balance subject to the specified boundary conditions.

2.3.1.1 Overall Design and Boundary Conditions

Component models in the context of this simulation model are essentially control volumes that enforce energy and mass conservation across their boundaries. To solve the system, the fundamental operation of the model need not be exposed to the loop solver, as long as certain conditions are met. Components obey the overriding system design wherein flow direction is predefined. Thus, a component model will have an inlet boundary condition at a specific point, or node, and an outlet "boundary condition" at another fixed point, or node. Components will often operate using other boundary conditions beyond the inlet state. As an example, consider an air cooled chiller component. In terms of physical connections, the chiller will have one inlet and one outlet fluid port. The chiller will also have a boundary condition of (entering) ambient air temperature and flow rate, and therefore an outlet air state as well. The solution algorithm need not understand this connection to the boundary to solve the system, instead relying solely on information transferred via the fluid inlet and outlet nodes. In another example, consider a water cooled chiller. This component will actually have four connections: an inlet and outlet on the chilled water loop, and another inlet and outlet for the condenser loop. The chiller component simulation will occur on each loop individually and the solution algorithm will be essentially unaware of the connections between the loops[5].

[5]An exception is at the very highest level of the simulation model, where interconnected loops are determined to optimize simulation order and monitor overall convergence, however this is irrelevant to the solution of a given loop.

As per this discussion, the component simulation model is a causal component in terms of a predefined flow direction, operating against an inlet boundary condition, any number of unspecified boundary conditions, and providing an outlet condition. This formulation is shown in Figure 2.4.

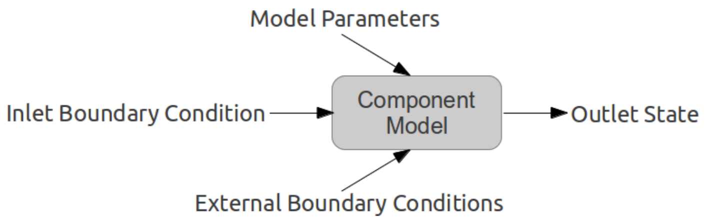


Figure 2.4: Generalized view of a simulation component model

The unspecified boundary conditions can include, but are not limited to the following list:

- ambient outdoor environment, as in the air-cooled chiller described above, or a cooling tower
- ambient zone conditions, such as an exposed pipe in a mechanical room or exposed ducting in the conditioned space
- ambient ground conditions, such as a buried pipe acting as an environmental heat exchanger
- a connection to another loop via a direct heat exchanger
- a connection to another loop via a refrigerant loop, such as the water-cooled chiller described above, or a heat pump

The component model has two responsibilities: ensuring continuity and providing a solution to the energy equation subject to the specified boundary conditions.

2.3.1.2 Continuity

Continuity in the system is ensured by providing a line of communication between the solver and the component. This is achieved by using a specialized interface called

SetComponentFlowRate. The name of this interface is a bit of a misnomer, as the interface does not directly set a flow rate, but rather allows components to issue flow requests, which are processed by the solution algorithms. This is described in detail in a later section (2.3.2), however it is noted here first as it is a key mechanism in ensuring that the component and system maintain continuity. This interface defines one required feature of all plant component models.

2.3.1.3 Energy

While continuity is ensured by utilizing the component interface, the energy equation is solved by individual component model formulations. Most models, though not all, are formulated as steady state. To provide feedback to the system, the goal of the component models is generally to solve a form of the sensible heat transfer equation:

$$\dot{q} = \dot{m} C_p \left(T_{out} - T_{in} \right) \tag{2.5}$$

This may seem like a nearly trivial concept, however further discussion provides insight into the complexity. There are four variables in equation (2.5) which have varying meaning based on the component type and solution state[6]. The specific heat, C_p, does vary based on temperature, however this effect is irrelevant to the current discussion, and the property may be treated as constant. The four relevant variables in equation (2.5) are discussed in Table 2.1.

Component models exist within a loop simulation inside a whole building energy simulation model, and the energy usage (electric power, for example) is also calculated by each model. Component efficiency is often dependent on entering conditions, which results in an energy usage dependence on operating conditions that is captured by the simulation model. Components may lodge their energy usage to output routines

[6]This is eluding to the fact that the solution algorithm is not an explicit solution, but rather an iterative implicit solution, and that the behavior of components may be different depending on the current solver state.

Table 2.1: Discussion of variables found in equation (2.5)

Variable	Comments
$\dot{q}$	The heat transfer rate may be a result of a component model calculation or a demand imposed on it as a request from the system solution algorithms. The difference depends on the current control strategy for this component.
$\dot{m}$	The mass flow rate in this equation is the result of a negotiation between the component model and the system model which ensures continuity while attempting to meet simulation demands. Flow may be requested by components based on design conditions or current demands.
T_{out}	The outlet temperature may be either a result of attempting to meet a heat transfer request/demand, or a desired outlet condition (set-point) for the component itself.
T_{in}	The inlet temperature is a fixed boundary condition at a given point in the simulation.

which can then aggregate these values to generate reports.

2.3.1.4 Model Types

The system simulation solver must be capable of managing a diverse set of component models. The diversity exists not only in the physical nature of the components as they exist on real loops, but also in the mathematical nature of the model. Handling this diversity in a robust way is a key feature of the ICE-B model.

If all component models in the simulation were of the same form, for example equation-fit representations, the system model may be reduced to a more simple configuration, where in the extreme case, the system model could be reduced to a fully graphical solution. However, this system model must handle both equation-fits and parameter estimation models, as well as numerical finite difference, and response factor model forms. The component model library in EnergyPlus is rich, allowing many typical and novel system configurations to be simulated, and utilizing the best mathematical form for each application. This is one reason for the selection of EnergyPlus as a basis for this model development.

Equation-Fit Models The term equation-fit models refers to models that rely on a (generally) small number of equations, with coefficients derived from manufacturer's data, experimental data, or otherwise estimated. The form of the equation need not represent any underlying physics of the physical object itself, as long as it suitably predicts the component response under given boundary conditions (including initial conditions for dynamic components). The equation could predict the response of any aspect of the component, with common models representing heat transfer or power as a function of entering fluid and other boundary conditions.

One of the air-cooled chiller component models in EnergyPlus is an example of an equation fit model. The air-cooled chiller model is a single inlet, single outlet component, which can be abstracted from the physical component to the generalized diagram of Figure 2.4. The component interacts with ambient conditions as an external boundary condition. The component has a number of parameters which may include physical design values such as design flow rate, capacity, and temperature conditions.

For this single inlet, single outlet component, the model will request flow on the loop-side to which it is connected.[7] The flow for the component may be requested based on design inputs. This is requested via the interface *SetComponentFlowRate*, which may adjust this value based on loop conditions and constraints. Once the operating mass flow rate is determined, it is joined with the entering fluid temperature to fully define an inlet boundary condition.

As an example, an EnergyPlus chiller performance model is described here. This model is derived from the BLAST simulation program (Blast Support Office, 1986).

Component performance is determined as follows, with nomenclature listed in

[7] In contrast, some components may be connected to multiple loops, such as a water cooled chiller that has both evaporator and condenser connections. See section 2.3.6.5 for details on coupled loop simulation.

Table 2.2: First a representative temperature difference, δ, is calculated based on entering and design conditions to be used in subsequent calculations:

Table 2.2: Nomenclature used in describing the equation fit component model

Variable	Description
$T_{in,condenser}$	A dynamic state variable representing the current condenser inlet temperature boundary condition (for this component model, it is the outdoor air dry bulb temperature)
$T_{in,condenser,design}$	A fixed user-input parameter for this chiller component
$\delta T_{rise,ratio}$	A fixed user-input correction factor for off-design operation
$T_{evap,out}$	A simulation setpoint for target chilled water supply temperature
$T_{evap,out,design}$	A fixed user-input parameter for design chilled water supply temperature
Q_{evap}	A water-side heat transfer rate for this chiller, typically prescribed by current control operation
Q_{design}	The design water-side heat transfer capacity for this chiller
$\alpha_i, \beta_i, \gamma_i$	User-input curve-fit parameters which define the component performance during off-design conditions

$$\delta = \frac{T_{in,condenser} - T_{in,condenser,design}}{\delta T_{rise,ratio}} - (T_{evap,out} - T_{evap,out,design}) \tag{2.6}$$

The available evaporator heat transfer capacity is then calculated as a curve fit (with coefficients generated from manufacturer's data):

$$Q_{frac} = \frac{Q_{evap}}{Q_{design}} = \alpha_1 + \alpha_2\delta + \alpha_3\delta^2 \tag{2.7}$$

The part load ratio is inferred from this variable using another curve fit:

$$PLR = \beta_1 + \beta_2 Q_{frac} + \beta_3 Q_{frac}^2 \tag{2.8}$$

The fraction of full load power is then calculated using yet another curve fit:

$$P_{frac,fullLoad} = \gamma_1 + \gamma_2 \left(PLR\right) + \gamma_3 \left(PLR\right)^2 \tag{2.9}$$

And finally the compressor energy use is calculated from these part loadings and a rated COP:

$$P = P_{frac,fullLoad} \times PLR \times \frac{Q_{frac}}{COP} \tag{2.10}$$

Equations 2.6 to 2.10 comprise the process of taking entering boundary conditions (temperature) and parameters (specified COP and design conditions) and resulting in energy use. This energy is then put on to the fluid loop via the component outlet state point (the fluid is colder on the component outlet, thus removing heat from the fluid). The compressor power can be reported to consider the energy use of this component. Variations of this particular model include variable COP which can also vary with entering and environmental conditions, and a component coupled to multiple loops to handle water-cooled chillers and heat recovery loops.

For this component model, the solver interacts with the component model by first providing a fixed entering condition: the entering temperature. This is calculated as the outlet response of upstream component models. The operating mass flow rate for the component is not specified directly, but is resolved through communication between the component and solver during the entire system solution. The solver also provides constraints to the component model including operating limits. The heat transfer for the component may be specified by control logic, however the heat transfer may be limited by the constraints imposed and component capacity. The outlet temperature may be a setpoint assigned again by control logic, however the ability to meet this request may also be limited by other constraints and component parameters. These boundary conditions and constraints are used by the component model with its own governing equations to determine a response and provide outlet conditions to be used downstream.

Parameter Estimation Models Parameter estimation models generally refer to models that use a mathematical representation that more closely represents the object physics than do equation-fit models. For example, instead of a polynomial representation of the heat transfer to entering fluid temperature relationship, this relationship may be modeled based on physical materials and geometry.

One example of such a model is the water-to-water heat pump parameter estimation model (Jin, 2002). The entire model description can be found in that source, although the compressor is shown as an example of the parameter estimation approach. The compressor mass flow rate is calculated, for a reciprocating compressor, as:

$$\dot{m} = \frac{PD}{\nu_{sec}} \left[1 + C_1 - C_1 \left(\frac{P_{dis}}{P_{suc}} \right)^{1/\gamma} \right] \tag{2.11}$$

The compressor power is calculated as:

$$\dot{W} = \frac{\gamma}{\gamma - 1} \dot{m} P_{suc} \nu_{suc} \left[\left(\frac{P_{dis}}{P_{suc}} \right)^{\frac{\gamma-1}{\gamma}} - 1 \right] \tag{2.12}$$

The form of equations (2.11) and (2.12) include a sense of physicality, which is why this model is termed a parameter estimation type. In these equations, there are a number of parameters which represent physical values of the compressor, as listed in Table 2.3.

Table 2.3: List of physical parameters found in equations (2.11) and (2.12)

Symbol	Description	Typical Units
PD	Compressor Piston Displacement	m^3/s
C_1	Compressor Clearance Factor	–
P_{dis}	Discharge Pressure	Pa
P_{suc}	Suction Pressure	Pa

The parameters in Table 2.3 are used to predict the compressor response. The heat pump model then includes models for the coil heat transfer. In this particular model the coils are effectiveness-based, however the coil model could be of any type. This parameter estimation model is itself a collection of sub-component-models of different formulation.

The system solver must be robust in handling diverse components without relying on a specific formulation type, including component models which are themselves collections of sub-component models. For this water-to-water heat pump component model, the solver provides multiple boundary conditions to close the component model's equation system on both hydronic loops to which it is attached. This requires proper simulation of multiple loops, including inter-loop communication of constraining conditions.

Numerical/Finite-Difference Models There are a few component models in the EnergyPlus library which utilize a finite difference form to solve the governing differential equations. One major application is in ground heat exchanger models, where the ground temperature is solved using a numerical grid while the fluid also passes through the plant. The component model no longer relies on the solution of a single differential equation to determine the fluid response, but instead relies on the solution of a large number of equations to be solved concurrently. This application is covered in great detail in the development of a new ground heat exchanger model (see Chapter 4). A transient energy balance is first established in the domain:

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T \tag{2.13}$$

This equation is discretized and solved at each node in the ground. In addition to the ground solution, the model acts as a component model on a hydronic loops. The hydronic component model consists of a single circuit of fluid that passes through the ground domain, from an inlet to an outlet. The ground may also contain multiple circuits that could exist on one or more hydronic loops. Thus the solver must be able to interact with this single domain object at any number of locations. At each location, the fluid circuit in the ground domain requires a well-defined entering condition, and properly applied constraints and control settings. The system solver handles this robustly because of a generalized development and proper interfacing between component models and the solver itself.

Response Factor Models A response factor approach is used in the solution of the transient wall conduction problems for zone simulation. An initial model of any form is used to calculate the response to impulses on the system of differential equations. The response is captured in the form of a series of dynamic coefficients. In the same way, a response factor approach is used to capture the response of a vertical borehole heat exchanger system Yavuzturk and Spitler (1999). The model uses superposition to aggregate the loads from the borehole history and a set of g-function response factors to determine the borehole temperature using the following equation:

$$T_{borehole} = T_{ground} + \sum_{i=1}^{n} \frac{Q_i - Q_{i-1}}{2\pi k} g\left(\frac{t_n - t_{i-1}}{t_s}, \frac{r_b}{H}\right) \tag{2.14}$$

The key feature of interest from this equation is the use of the series of response factors g_i, which represent a transient response to a series of historical conditions in the borehole system. The system simulation model must be able to handle dynamic simulation models along with a variety of other simulation model types, including steady state or quasi-steady state. Transient component models used with this solver are required to handle the iterative nature of the solution algorithm, and provide continuity and energy balances through changes in time step size.

Connector Component Models Pipes may be modeled as either heat transfer components or simple flow connector components. Flow connector components are idealized, adiabatic, pipes. There are no physical parameters required to define their operation, and no external boundary conditions in relation to Figure 2.4. When this component is simulated, it simply takes the inlet condition, and passes it to the outlet state point ($T_{exit} = T_{inlet}$). No flow requests are made and the component transfers no heat to the loop. Although this component is conceptually trivial, it is a required component for the generalized solver, in which every branch should have at least one component, no matter how simple or complex. While this component is trivial, it must still provide continuity and energy balances like all other component models.

2.3.1.5 Summary

The EnergyPlus central plant simulation component model library is diverse in terms of the physical objects to be simulated, and also the types of formulations used to solve the component's governing equations.

Consider a contrasting scenario, in which all component models were of the same form. In this case, the system simulation model would have a pre-determined understanding of the interconnections between all the components available for simulation. This would make simulation of such components easier, and could even reduce the system simulation model to a single matrix solver, for example, if all coefficients were known for all components in a single form.

There are many applications where the same form does not fit between components. A parameter estimation procedure works well for components that rely on a small set of governing equations, and especially well for steady state solutions. For transient solutions, such as a ground heat exchanger, the number of equations to be solved gets larger, and the interaction between simulation features becomes stronger. As such, a more general approach is utilized in these cases, either a response factor method or a numerical/finite difference form. As such, the solver for the new model is able to handle any component model type by hiding the underlying component-model solution in the component itself, and communicating via specially designed interfaces. The solver makes certain things available to component models, such as historical data at points in the system, however the physical calculation of component response is independent of the system solver itself.

2.3.2 Flow Network Solution

The new model proposed here, and termed the Improved Continuity & Energy-Balance (ICE-B) system model was designed to replace the existing solver described by Fisher et al. (1999*a*). Both the existing and proposed models attempt to provide the solution to the system of component-interconnected state points, while trying to meet loop demands and ensure system convergence is attained. The ICE-B model uses a predictor/corrector approach where a flow request is initiated by a component, and represents the predictor step in this logic.

This initial discussion describes an underlying feature of the system solver: the use of a flow request mechanism along with two-way communication between components and the system solver, and the iterative coupling between them.

2.3.2.1 Flow Request: A Definition

The flow request is an extra variable used to describe state points in the system. This is unrelated to a thermodynamic state point, instead acting as a metadatum, used to provide the simulation with additional information. The flow request is used by the solver to resolve overall loop and individual leg flow rates.

2.3.2.2 Flow Request: Methodology

A flow request is a notice issued by a component model and passed to the system solver. The solver takes this value, logs it, and makes decisions regarding network flow conditions. The flow request mechanism is a key feature, ensuring that the simulation provides continuity and a stable iterative solution algorithm.

Table 2.4: Simplified description of how component models calculate a flow request

Component Description	Flow Request Overview
Constant Flow	This component will make a request based on a design flow rate, which can be specified by a user, or auto-sized based on demand conditions encountered during sizing calculations.
Variable Flow	Variable flow components will calculate a flow request in order to meet an outlet temperature setpoint and also possibly to meet a certain demand from the loop.

The flow request is made by a component model based on a number of conditions. Some examples are listed in Table 2.4. The flow request lodged by a component model is not necessarily the flow rate at which the component will operate. It is instead an indicator to the system of the component's desired (perhaps optimal) state. The value of the flow request is logged by the *SetComponentFlowRate* routine as a part of the state point definition to be retrieved later. As previously discussed, this mechanism is a two-way communicative interface between the system solver and the component model. The component model is able to try to meet demands and affect the system solution by lodging requests, and the solver is able to adjust the actual component flow rate and attempt to meet these requests. This reduces the responsibility of the component model by taking away overhead related to the state of the simulation. In the existing simulation solver, component models checked the status of various parts of the simulation to determine a suitable operating point. This ill-defined protocol could not guarantee that both continuity and energy balances would be enforced at the component and loop levels. With the new interface, the component always follows a set pattern during every step of the simulation:

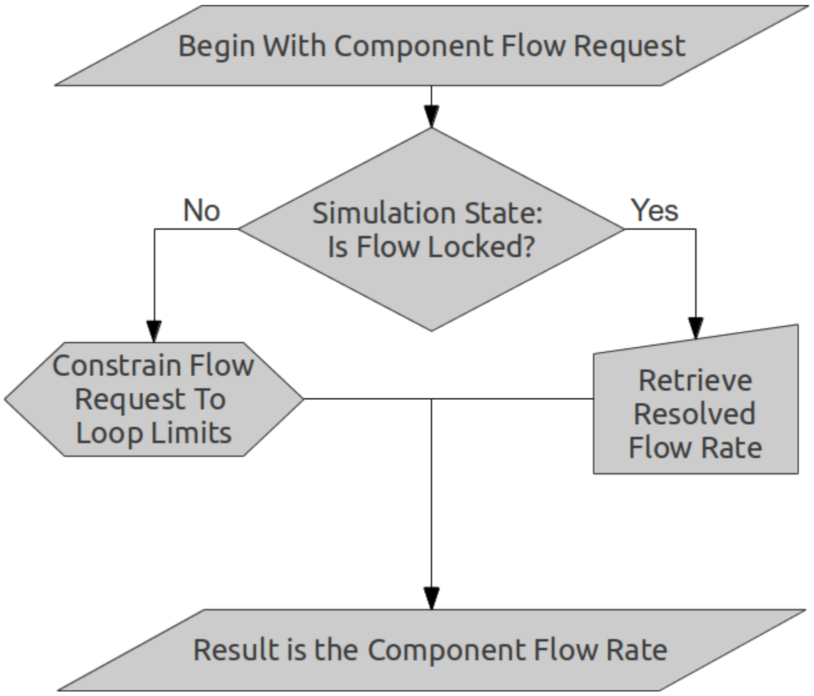
1. Calculate/determine a desired flow condition
2. Use the interface to exchange information with the solver (The solver will always return a suitable, *stable*, condition)
3. Operate under this possibly adjusted condition, even if unable to meet demand

The communication between the solver and component models provides a stable simulation environment for solving the system of state points. The flow request logic has additional important features: the ability to directly handle variable flow operation, perform diagnostics and monitor convergence. The request is essentially the predictor step in the predictor/corrector logic of the system simulation model, and may not be the final resulting flow rate. This flow request logic is also a key part of allowing for variable flow components to be simulated, monitoring system convergence, and providing operation diagnostics. The flow requests are carried through the simulation as accompanying metadata to the thermodynamic state point data, and as such, they can be compared to the final (converged) state of the system. This can feedback useful information about flow-starved legs of the network and therefore be useful in diagnosing other simulation results such as zones which are out of control. System convergence, which is described in section 2.3.6.4, is monitored by comparing the variation of flow requests lodged at specific points in the system throughout simulation iterations.

2.3.2.3 System-Component Flow Interface

Component models determine a flow request, call the routine *SetComponentFlowRate*, and then use the resulting, possibly adjusted, flow rate for subsequent calculations and reporting. This hides many overhead and bookkeeping issues from the component which minimizes the possibility of bugs in developing component models.

The solver can be in one of two states: either "unlocked" or "locked", as shown in Figure 2.5. These states correspond to the simulation logic for the predictor and corrector steps, respectively.


Begin With Component Flow Request
Simulation State: Is Flow Locked?
No
Yes
Constrain Flow Request To Loop Limits
Retrieve Resolved Flow Rate
Result is the Component Flow Rate
Figure 2.5: Basic logic of the *SetComponentFlowRate* routine

In an unlocked state, continuity is not enforced around the loop. Instead loop limitations are enforced, including:

- Specified maximum flow limits on the loop (presumably based on hardware limits or similar)
- Pumping limitations
- Flow restriction (essentially a high pressure drop situation reducing the maximum flow available)

In this way, components are provided a realistic estimate of available flow in the system. This is a loose simulation state that again does not ensure continuity throughout the flow network. Components will take this loose estimate and predict an impact on the loop. After the loop is loosely simulated, the flow rate around the loop is fully resolved to enforce continuity while attempting to meet flow requests (see sections 2.3.2.4 and 2.3.2.5), and the flow is locked down to begin the corrector step. Under this locked state, components will again attempt to request a flow rate (independent of the simulation state). With the flow locked, the solver interface always returns the resolved flow for that leg of the system to ensure stability and continuity in the simulation. The component must then take this flow rate and attempt to meet demand and operate within the specified control strategies. The existing model did not have a specifically designed interface, instead relying on each component model to check various simulation states and use global locked states to perform the simulation.

2.3.2.4 System Level Flow Resolution

In a physical system, the total loop flow rate is reached in a balance between the pressure head addition of the pumping system and the pressure drop of the remainder of the system, resulting in an operating point on the pump curve(s), which could certainly involve variable flow controls throughout the system. In the new simulation environment, the flow rate is not driven by pressure, and the total flow rate must be calculated in a different manner.

In the new simulation, the total loop flow rate is determined based on pumping capability and loop limitations, correlating with the physical system. To determine this loop flow rate, the total maximum pump capacity must be determined. This is achieved by querying the pumps individually to update their maximum capability. The pumps may have varying abilities based on scheduling limitations (some pumps only available at off-peak time, for example). Each pump will provide the system solver a maximum available flow rate. Since series pumping is not implemented for a single loop-side, adding up all loop-side pumps provides a total system pumping capacity. The possible pump configurations are shown in Figure 2.6.

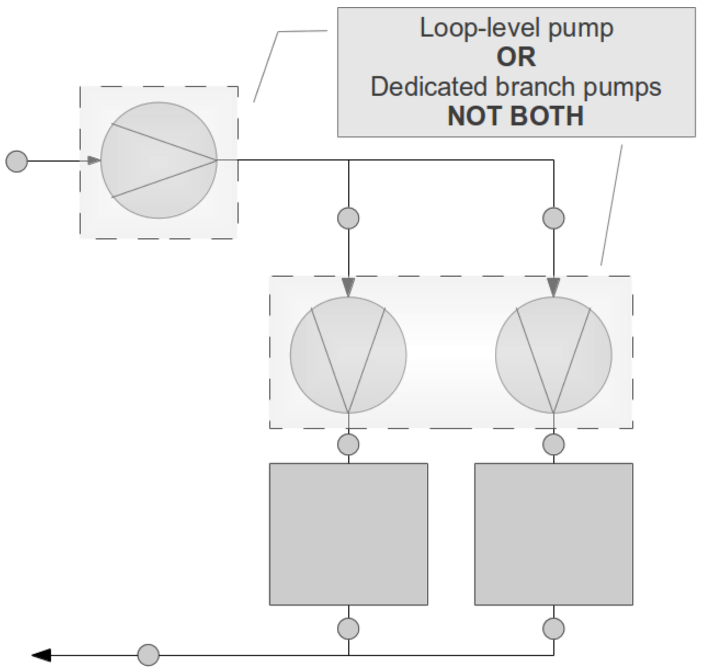


Figure 2.6: Allowable pump placement configurations for a single loop-side

The total pumping capacity may exceed other loop limitations. These limitations can include:

1. explicitly specified maximum loop flow rates
2. explicitly specified maximum component flow rates which result in a constrained overall flow
3. a restricted flow condition encountered during a simulation in which components in the loop can only handle a certain maximum flow

The first two of these are encountered during solver initialization, and limitations can be imposed early. The third is a dynamic issue that must be considered at every iteration as the system converges. This is analogous in a physical system to a loop which does not have a 3-way control valve with bypass, but instead controls flow through demand coils with 2-way control valves. As the 2-way valves are closed, the pressure drop increases, and the pumps respond with a reduced flow. The simulation model responds similarly, but the restricted flow condition is not driven explicitly by a high pressure drop.

The final piece of the puzzle is how to determine an initial guess at loop flow request for a given iteration. This is calculated based on the most recent individual component flow requests. The maximum request by any component on a branch becomes the representative flow request for the entire branch. If the loop-side contains parallel branches, the branch flow requests for each are summed in parallel and then compared to the remaining series portions of the loop. The maximum flow rate is selected, noting that some components such as variable speed pumps will not themselves issue a flow request, instead relying on the remaining components on each leg to initiate flow requests.

The logic used in determining the total loop flow rate is shown in Figure 2.7. Analogous actions of the physical system's components are also shown in the Figure. Once the total loop flow rate is obtained, it is enforced on this loop-side, regardless of whether it is suitable for meeting the current demand and flow request. It is a stable value using component and pumping limitations to ensure the system maintains control (simulation semantic control, not necessarily setpoint control).

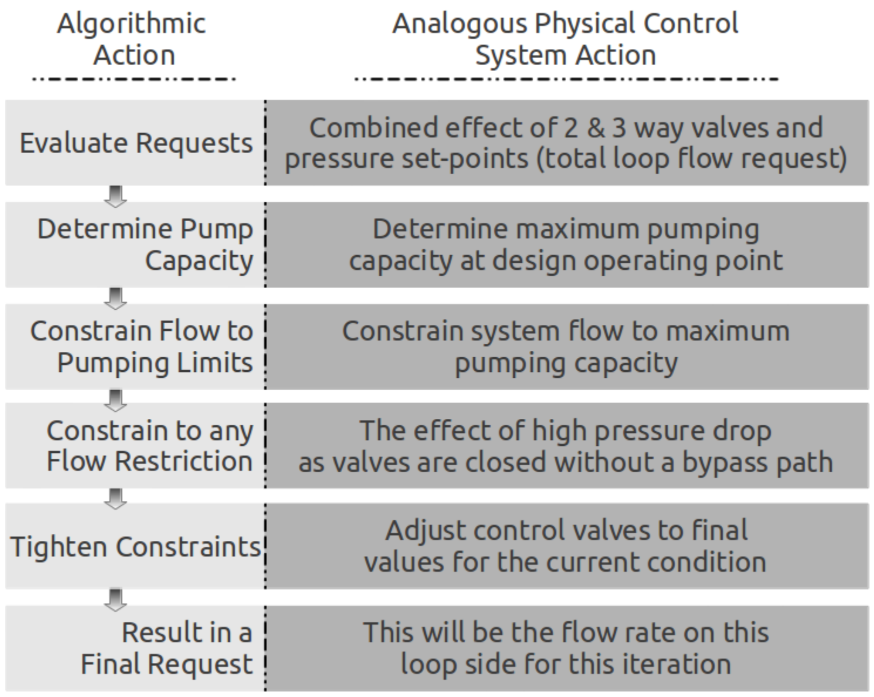

Algorithmic Action
Analogous Physical Control System Action
Evaluate Requests
Combined effect of 2 & 3 way valves and pressure set-points (total loop flow request)
Determine Pump Capacity
Determine maximum pumping capacity at design operating point
Constrain Flow to Pumping Limits
Constrain system flow to maximum pumping capacity
Constrain to any Flow Restriction
The effect of high pressure drop as valves are closed without a bypass path
Tighten Constraints
Adjust control valves to final values for the current condition
Result in a Final Request
This will be the flow rate on this loop side for this iteration

Figure 2.7: Process of determining a total loop flow rate

The simulation then proceeds for this loop-side to simulate individual component models and distribute flow to parallel legs. The parallel distribution approach is described in the next section. This entire process is iterative with the ultimate goal that the flow request and components will either converge on a total flow rate and flow distribution that satisfies pump capacity and restrictions, or the loop will be unable to meet all demands or requests. This situation occurs in a real system when flow control valves have lost authority due to insufficient flow (low pressure drop) in the system.

2.3.2.5 Parallel Leg Flow Resolution

As described in section 2.1.3, a loop-side may contain parallel legs in between a set of series inlet and outlet legs. If it does not contain this parallel set, then the total loop flow rate calculation described in the previous section is sufficient to resolve the flow through the series system, and satisfy continuity. The current section describes the resolution of the total loop flow rate for parallel component paths.

In a controlled physical flow-network, the calculation of a pressure network allows the computation of uncontrolled loop flow rates. Without a pressure network calculation, the proposed simulation model algorithms assume that the control valves on the system close instantly in an ideal fashion to regain authority over the network, and distribute remaining flow according to operating priorities. These operating priorities are established by specifying the type of flow control associated with the component. Since the total flow through the entire network is enforced based on previous calculations, this step can be understood as determining the ratio of the total loop-side flow rate through each branch. This can be demonstrated first by example.

Consider the parallel path network shown in Figure 2.8. In this figure, three components are shown with different flow control labels. The “Active” component generally represents a piece of equipment with a two or three way control valve or

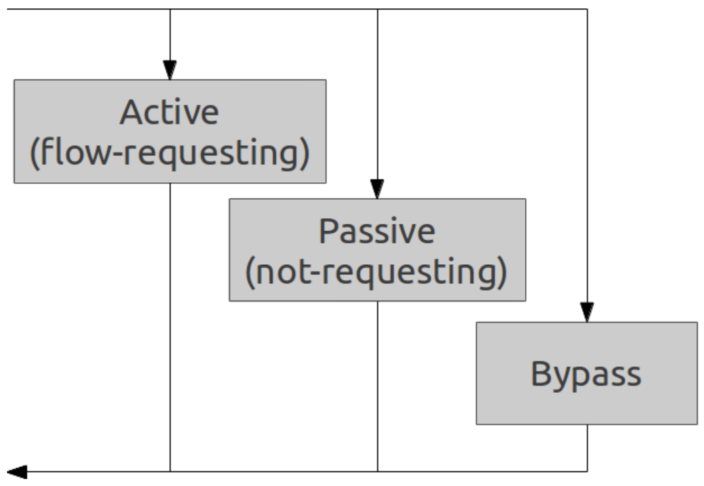


Figure 2.8: A simple parallel path showing components of various control classes

a dedicated pump and a high priority in receiving flow, such as a chiller or cooling coil. “Passive” components do not have a means of controlling their flow, but are dependent on the current pump/system configuration to determine flow. Ground heat exchangers are often configured this way. These components have a lower flow priority than “Active” components, as they may not be as closely coupled to the loop supervisory controls as “Active” components. When flow is available, “Passive” components will operate like “Active” components, however in flow-starved or excess-flow cases, “Passive” components have a lower priority in receiving the flow they request. “Bypasses” are parallel legs that consist only of a pipe component. Use of a “Bypass” component implies that at least one parallel component on the loop-side is controlled by 3-way valves, and the “Bypass” will close down as needed to ensure flow is distributed to other controlled components. Each of these component types are described in terms of their flow request and flow distribution design in Table 2.5.

The flow resolution procedure follows the following logical progression:

1. Attempt to provide all active branches with their requested flow rate. This corresponds to adjusting control valve positions for all active components to ensure they get the flow they request.

    - If the network is now flow-starved because of a lack of pumping capacity,

Table 2.5: Brief overview of component control types in relation to flow requesting and flow distribution

Type	Flow Requesting	Flow Distribution
Active	Requests flow based on current demands and any other component model calculations.	These components have the highest priority in receiving the actual request, thus the least likely to be starved or overfed with flow.
Passive	Requests flow based on design constraints and component model calculations.	While the flow requests are lodged in determining a total loop flow request, these requests are considered only after active requests have been met during parallel flow resolution.
Bypass	Does not request flow.	In the case of excess flow (more flow than requested), bypass legs, when included in the loop topology, will take excess flow to allow active (and passive) components to receive their requested flow.

control priority is given to components in the order specified in the controls design in the simulation input.

2. Attempt to provide passive branches with their requested flow. This corresponds to adjusting control valves for passive branches, without affecting the flow control for the already adjusted active components. It is then assumed that the control valves automatically and ideally adjust to the changed pressure distribution in the network.

    - Passive branches are given their requested flow in order of appearance in the simulation input specification. If there is not sufficient flow for a component, it is assumed that the control valve ideally closes down to regain authority over the flow so that further flow distribution can be handled.

3. Distribute excess flow through the bypass components, if they exist.

    - Bypass branches can inherently handle any flow rate, so if any bypass exists on the loop-side, it is impossible for excess flow to exist beyond this point in the control logic.

4a. In the case that excess flow exists without a bypass path and without a loop pressure drop calculation in place, distribute the excess flow.

- First the passive components accept excess flow by opening their control valves first, thus providing a control preference to the other active components.
- If there is still any excess flow to be distributed, it is then distributed to the active components.
- The parallel legs will never encounter a total loop flow greater than the maximum capacity because the maximum loop capacity is included in the total loop flow constraints.

4b. In the case that excess flow exists without a bypass path, however a pressure drop calculation is in place, adjust pumping capacity using the pressure drop.

- For the current iteration, the flow is distributed according to step 4a. However, in the following iteration, the pressure drop and system flow rate are used to calculate a representative system curve, which is used in conjunction with a pump curve to resolve to a restricted operating point.
- The standard approach used in the proposed model does not require pressure drop information. These supplementary pressure drop calculations are described in section 2.3.3.

At this point, the flow has been resolved using an algorithmic approach instead of a traditional pressure network solution[8]. As an example, consider a physical system that is analogous to the simplified network in Figure 2.8. In a physical system, there will be valves placed on each leg that work in conjunction with controllers to ensure

[8]If pressure drop calculations were utilized, they are used in resolving the total loop flow rate, not resolving individual parallel path flow rates.

fluid is passing through the appropriate legs. At high demand, the active component, which could be a chiller, may need the full capacity, so the other valves are shut down and the active branch is fully open. As demand reduces, a secondary heat exchanger may be utilized to handle some demand, this being the passive branch. In cases where the chiller or heat exchanger flow demand is lower than the minimum pump flow rate, the bypass leg is opened to allow smooth operation of the system. In this simulation, the valves are essentially an idealized, inherent aspect of the logical flow resolution technique. Instead of measuring pressure drops and adjusting valve positions, flow requests are processed to determine a resulting network flow solution.

2.3.3 EnergyPlus Pump Models

The models used to simulate pumps are not significantly different in the ICE-B model than the existing model. The main difference between the two occurs at the interface between the system model and the pumps, not within the pumps themselves. The system solver has a special state that occurs at the beginning of the predictor-step which is used solely for querying the pumps located on the current loop-side. This step allows the solver to calculate the maximum pump flow capacity at the current time, which could vary based on pump scheduling and availability. This information along with many other variables are used to determine the current loop operating point (section 2.3.2.4). Once the maximum pump capacity has been determined and used in determining the loop flow rate, the pumps are treated like other component models: as control volumes which must themselves enforce continuity and energy balances over their mathematical boundaries.

While pump model details are *similar* between the existing and proposed EnergyPlus simulation models, the pump simulation methodology is much *different* than in those models that are based on a pressure network solution. Section 2.3.3.1 provides a description of the standard modeling approach used in this system simulation. Section 2.3.3.2 provides a description of an alternative pressure-based modeling approach, followed by supplementary details on the approach used in the proposed model.

2.3.3.1 Non Pressure-based Modeling

The solver has the ability to provide the pump component model with a desired flow condition, however the pumps themselves can control their own desired flow condition based on scheduling or other control management operations. The pump will take this information and calculate a flow request, which is sent to the *SetComponentFlowRate* interface. This routine logs the request, and makes appropriate adjustments to obey loop flow constraints, and returns back a suitable flow rate, which the pump must use. To allow for more stability with diverse applications, the proposed solver now stores pump requests at a higher level instead of relying on the pumps themselves to track this information. This improved the possibility of placing pumps in diverse configurations on the loop.

Non pressure-based pump modeling includes special features for two cases: constant vs. variable speed pumps and banks of headered pumps.

Constant vs. Variable Pumping The constant speed and variable speed pump models do not rely on pump curves directly. Pumps have the ability to induce flow in the network by issuing flow requests along with the rest of the components, as described in section 2.3.2.2. This is where the key difference between constant and variable speed pumping models is found. Constant speed pump models will attempt to provide flow at a design or otherwise specified rate, independent of the state of the simulation and components. Variable speed pump models do not explicitly make flow requests, but instead remain available to providing flow as desired by other components on the loop. Variable speed pumps can be thought of as followers, not leaders, in determining loop flow conditions. Of course without the pumps in place there wouldn't be any flow, just like in a real system. (While this situation would be a trivial exercise, the lack of any pumps would naturally be recognized in the loop flow calculations in section 2.3.2.4.) Capturing the constant and variable speed pump operation in this fashion is idealized, however more advanced capabilities have been implemented to perform more detailed pressure-based studies (section 2.3.3.2).

Pump Banks In many systems, banks of pumps are headered to provide either maintenance or safety redundancy, or to provide the ability to stagger operation and optimize on-hours for individual pumps. These are modeled in the current simulation, but as a "black box." At the interface with the system solution, pump banks appear as a single pump (either constant or variable flow). The entire pump bank consists of a single inlet and single outlet, and provides a single flow request to the system solver. Once flow is resolved, however, the pump bank models post-process the flow along with pumping configuration to determine the number of pumps running in the pump bank. This allows a suitable representation of the part-load operation and energy use.

2.3.3.2 Pressure-based Modeling

Both the component and system simulation models have been described in a pressure-less fashion, with an emphasis on the suitability of this methodology in a whole building energy simulation environment. In these simulations, the efforts of pump curve fitting, system curve modeling, and solution of a full pressure network may not be justified for evaluating the energy use of the plant. However, the simulation model has been extended to provide a layer of pressure calculations to allow a more accurate representation of energy use, or flow constraining operation to be captured, without moving to a full pressure-based flow network solution. This section describes the pressure-based methodology calculations and limitations, including the pump model formulation for cases where the pump also includes pressure information. The first step is to calculate a pressure drop for the entire loop.

Loop Pressure Drop: For loop pressure drop calculations, each flow leg (branch) of the system can have a representative pressure drop. These individual branch pressure drops are lumped together to achieve a total loop pressure drop for further calculations. The pressure drop on any branch may be calculated using one of two options: a predefined pressure formulation or a generalized function of mass flow rate. These two forms are summarized in the following table:

Formulation	Expression	Notes
Predefined	$\Delta P = \left(f\frac{L}{D} + K\right)\frac{\rho V^2}{2}$	Major and minor losses
Generalized	$\Delta P = f\left(\dot{m}\right)$	Any other formulation

Note that the predefined formulation accounts for the possibility of both major and minor losses for a given branch of the system, using both the friction factor f and the minor loss coefficient K. The generalized formulation can be any of many different forms which are built into the simulation shell including linear, quadratic, cubic, quartic, logarithmic, and others. Any univariate functional form could be utilized for a specialized case.

The individual branch pressure drops are then combined into a total loop pressure drop for the current flow condition. The maximum pressure drop of the legs on any parallel set is the representative pressure drop for the entire parallel set. The overall pressure drop calculation for an entire loop (both demand and supply loop-sides) is calculated as:

$$\Delta P_{loop} = \left[\Delta P_{loop-side}\right]_{demand} + \left[\Delta P_{loop-side}\right]_{supply} \tag{2.15}$$

$$\Delta P_{loop-side} = \Delta P_{inlet-branch} + \Delta P_{parallel} + \Delta P_{outlet-branch} \tag{2.16}$$

In subsequent calculations, the total loop pressure drop is used, with the individual terms being used only to calculate this total loop pressure drop. This approach results in a flexible model, however also reveals a requirement on the usage:

Flexible: Pressure drop information is not required for every branch of the entire system.

Limitation: The lumped nature of the pressure model does not perform pressure-based parallel path flow resolution, instead relying on and working with the logic (flow-request) based flow resolution model.

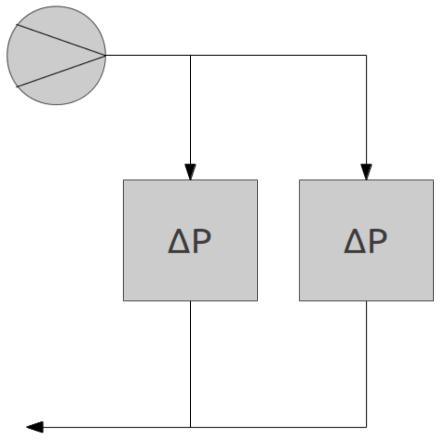
Requirement: Every flow path through the system must have at least one pressure drop component, though the placement is flexible. Consider a loop-side with two parallel components. Three obvious cases are possible:

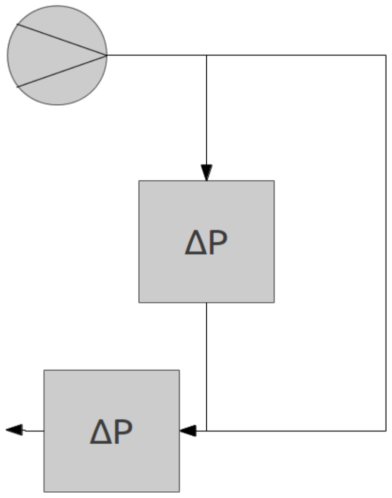
- Figure 2.9a shows pressure drop components on each parallel leg, thus every path through the system will have a representative pressure drop.
- Figure 2.9b shows that one parallel component does not have a pressure drop component, however the outlet leg does, so every path will still have at least one pressure drop object.
- Figure 2.9c shows an invalid case where the second leg of the system does not have a pressure drop component, which is invalid.

The pressure drop in the parallel system is not used for parallel flow resolution, instead relying on the algorithmic, flow request-based, flow resolution model. To accommodate pressure balancing in these parallel systems, the pressure model includes "valves" inherently placed at the outlet of each parallel branch, as shown in Figure 2.10. The highest pressure drop in the parallel branch set is determined and used as the pressure drop for the set of parallel components:

$$\Delta P_{parallel} = \max_{i=1}^{NPP} \Delta P_{path,i} \tag{2.17}$$

$$\text{Where: } NPP = \#ParallelPaths \tag{2.18}$$


(a) Case A - Valid


(b) Case B - Valid

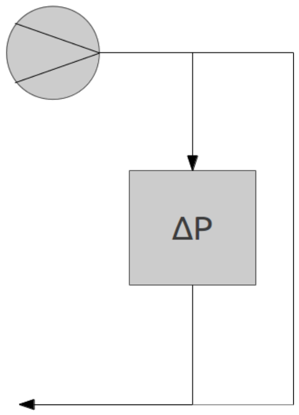

(c) Case C - Invalid

Figure 2.9: Possible pressure drop object placement configurations for a given loop-side

The remainder of the parallel branches will all report at this same pressure drop by using a “valve” to increase the pressure drop to match.

$$\Delta P_{valve,i} = \Delta P_{parallel} - \Delta P_{path,i} \tag{2.19}$$

Note that since the pressure drop is not used in resolving parallel flow, this will not have an effect on the final mass flow rate through the branches. This is an idealized system that can meet the resolved flow system using an arbitrary pressure drop. Stated differently, the pressure drop, as described in this section, is predominantly just a value that is post-processed from the main simulation algorithms. While these individual branch/valve pressure drops are not used in flow calculations, the final pressure drop calculated for the entire loop is useful for further loop-level calculations.

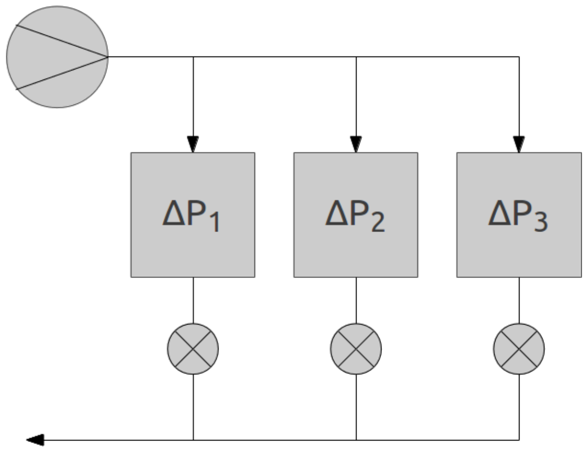


Figure 2.10: Pressure drop with idealized valves and control

Pressure-based Pumping: Stage 1 One possibility for using the loop-level pressure drop is to better predict pumping energy. With the standard pressure-less pump model, the energy is a function of (at most) the ratio of current flow rate to design flow rate. With a loop-level pressure drop, the resolved system flow rate can be used to better predict the pump energy added to the fluid, using the simple flow expression:

$$W = (\Delta P_{loop})\, Q \tag{2.20}$$

This improves the estimate of pumping energy without requiring a detailed pressure network solution which would then require a substantially larger amount of input parameters.

Pressure-based Pumping: Stage 2 A second possibility for using the loop-level pressure drop involves the use of a pressure-flow pump curve to allow a constant speed pump to "ride the pump curve". The pump is modeled as a dimensionless curve-fit to manufacturer's pump data in a fourth-order (quadrinomial) form:

$$\psi = \sum_{i=0}^{4} C_i \phi^i \tag{2.21}$$

Where:

$$\begin{aligned}
\Delta P &= \rho \left(\frac{N}{60}\right)^2 D^2 \psi \\
\dot{m} &= \rho \frac{N}{60} D^3 \phi \\
N &= \text{Rotation speed} \\
D &= \text{Impeller diameter}
\end{aligned}$$

This pump curve is used along with iteration to resolve the system into a pressure-based operating point. During the initial step, no pressure data is available, so the simulation uses the standard flow-request logic to determine a system flow rate. At the end of this step, the total loop pressure drop is calculated. In subsequent iterations, the latest pressure drop is used to resolve the pump to a pressure-based operating point on its curve. During each iteration, a new total loop flow is determined based on total loop pressure drop, while the flow resolution to parallel paths is independent of these pressure-based calculations.

Pressure-based Pumping: Stage 3 A third possibility extends the constant speed pump curve to allow for variable speed pumping. In this case additional control parameters are placed in the simulation input specification and at each iteration, the pump model utilizes lagged flow and pressure drop information, along with the control parameters, to determine whether to adjust pump speed to then adjust loop flow conditions.

The pressure-based enhancements were performed as a collaborative effort along with Phalak (2011) as part of the new solver development project.

2.3.3.3 Summary and Analysis

In the new simulation model, pumps do not follow a behavior that may be expected when contrasted with pump modeling in pressure-based flow networks. The technique utilized here provides certain advantages and other limitations, which can be summarized:

Advantages of this Pump Modeling Strategy

- Fits with the pressure-less system simulation model
- Minimal input requirements
- Logical structure helps ensure robustness

Missing Features

- The base mode for resolving loop flow rate is unrelated to the pressure drop characteristics of the loop. Section 2.3.3.2 describes the addition of pressure attributes into the solver, however even with those pressure attributes in place, a combined solution between system operating point *and* parallel flow distribution is not implemented.
- For the base, pressure-less, simulation mode, pumps could be enhanced with an advanced relationship between efficiency and flow/part loading.

2.3.4 Controls

Component models are assigned a "flow control type" to be used during loop flow and parallel flow resolution. This "flow control type" is not directly related to the way supervisory controls interact with the component to meet setpoints. Rather the "flow control type" defines the flow priority of different components during flow resolution. Detail was provided in section 2.3.2.5.

The current section describes the setpoint control classification of components and how they operate to provide control to the system, as well as introducing the calculations performed to determine loop demand via the system's governing equation.

The proposed loop solver is generalized to be capable of solving for any loop-side in a system of hydronic loops. The loop-side form to be solved is discussed in section 2.1.3, and shown specifically in Figure 2.3. The loop solver is responsible for taking an entering boundary condition, and solving the system of component models to result in a state point distribution and overall loop-side response.

A loop-side may be controlled or uncontrolled. An uncontrolled loop-side can be considered as a typical demand side. In a chilled water system, the demand side will consist of chilled water coils. Though these coils may be controlled with valves in order to meet an air-side demand, they are not generally controlling to meet a target water outlet setpoint.

In contrast, a controlled loop-side can be considered as a typical supply side. In a chilled water system, the supply side may consist of a chiller and heat exchangers. These components are typically controlling to meet a target water setpoint, either at their local outlet or the overall chilled water supply temperature.

The solver can handle any loop-side, whether controlled or uncontrolled, in the same manner. In fact, corresponding demand and supply sides can be controlled to different points, assuming equipment is in place and controls are properly defined. However, for discussion, the supply side will generally be a controlled loop-side, while the demand loop-side is uncontrolled.

2.3.4.1 Governing Equation

In general, the loop solver is trying to control the loop by solving for the loop demand:

$$\dot{q}_{demand} = \dot{m} C_p \lambda \left(T_{out,setpoint} - T_{in} \right) + \sum \dot{q}_{sources} \tag{2.22}$$

Where:

$$\lambda = \begin{cases} 1 & \text{for heating operation} \\ -1 & \text{for cooling operation} \end{cases} \tag{2.23}$$

The first term on the right hand side of equation (2.22) reflects that the fluid entering the loop must be brought to a setpoint temperature, which will require some heat transfer. This demand can be pre-calculated once a loop flow rate has been determined as described in section 2.3.2.4. The outlet setpoint is not a single component outlet, but instead it is generally the loop outlet, as shown in Figure 2.11.

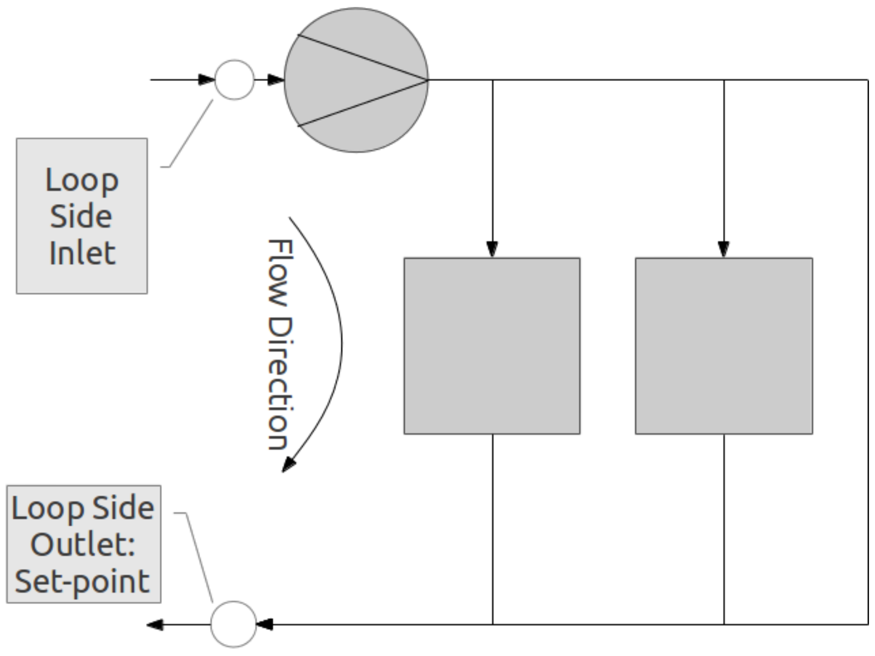


Figure 2.11: Typical location of loop setpoint temperature on a loop

The second term in equation (2.22) represents other sources on the loop, which are not known at the beginning of a solution step. These sources may be uncontrolled components, component-setpoint components, or other sources, but to the solver, they appear as discrete sources. As will be described in following sections, the interaction between the flow-wise solution and the order of components in the system will prove to be important in defining the system's ability to solve the loop to a controlled state. Once a loop demand is known, it may be dispatched to controlled components based on different control strategies.

2.3.4.2 Control Classes

Component control can be classified into four categories:

- Uncontrolled
- Component Setpoint
- Loop Setpoint
  - Constant Flow Components
  - Variable Flow Components
- (EMS/User-Defined)[9]

This classification is based on the interaction between the loop solver load dispatch algorithms and the resulting impact the component makes on the loop.

Uncontrolled operation decouples the component's governing heat transfer equation from the loop solver solution. The effect of this is similar to an addition to the independent source term in the governing solver heat transfer equation (2.22). The solver has no mechanism of utilizing these components in meeting control strategies, and must monitor the response of these components to ensure the effect is handled

[9]These components must abide by the rules of the solver-component interface and submit flow requests in the same fashion as other components, however the method by which these components affect the loop demand may differ based upon any number of user-defined variations.

in the remainder of the solution. Typical uncontrolled equipment consists of chilled water coils and pipes that include heat transfer effects. Although the chilled water coils are controlled on the air-side, they are typically not attempting to meet a chilled water target setpoint.

When a component is operating under *component setpoint control*, it ignores loop-level controls and tries to meet a temperature setpoint specified at the component outlet node. The most prominent example of a component setpoint object is a chiller used in ice (thermal) storage. The chiller may have a target outlet setpoint for freezing mode and another for when the thermal storage is discharging. The key point is that these components are not trying to meet a loop demand, but rather their own outlet setpoint temperature. Generally the solution of the component governing equation falls into a familiar form:

$$\dot{q} = \dot{m} C_p \lambda \left(T_{out,setpoint} - T_{in} \right) \tag{2.24}$$

At any point in the simulation, the outlet temperature setpoint is a *known* value. It may be constant, scheduled, or determined based on other controls formulations, however to the component model, this is a known value. The inlet temperature is a known entering boundary condition calculated from upstream components and provided by the system solution algorithm. The specific heat will vary with temperature, but is explicitly calculated—it is independent of the current solution. Thus there are two variables to be solved from equation (2.24): $\dot{q}$ and $\dot{m}$. The mass flow rate of the component is dependent on design specifications, pumping capabilities and flow restrictions on the loop. The heat transfer of the component varies with the inlet boundary condition, the resolved flow rate, and the component's functional relationship between heat transfer, flow rate, and temperature difference.

The iterative nature of the loop solver is applied to this component in the following fashion to provide a solution:

1. The demand (heat transfer rate) applied to the component is predicted initially based on a user-specified design flow rate and entering boundary conditions using equation (2.24).
2. The component will be simulated and may request a flow different from the design value in order to meet the demand applied to it.
3. The flow in the loop is resolved according to logic described in previous sections.
4. The component is then simulated with a fixed flow rate and will attempt to meet the outlet setpoint independent of other conditions on the loop.

Note that the component may not hit the outlet setpoint for a number of reasons, including out-of-capacity conditions and adverse flow conditions.

The effect of *component setpoint* controlled components on the loop itself is similar to uncontrolled operation: a heat transfer rate is added to the loop which the solver has no means of controlling, equivalent to an addition to the source term in equation (2.22). Thus the solver must monitor the effect these components have on the loop in order to ensure that broader control strategies can be applied properly.

Loop setpoint controlled components operate closely with the loop solver to meet the broad needs of the loop. Uncontrolled and component setpoint operation components are treated as source term contributions to the solver, as the solver cannot utilize these components in overall loop control. In contrast, loop setpoint components are tightly controlled by the solver to control a loop setpoint temperature. Loop setpoint components are governed by the following equation:

$$\dot{q}_{demand} = \dot{m} C_p \lambda \left(T_{out} - T_{in} \right) \tag{2.25}$$

While this governing equation of loop setpoint components may look similar to the component setpoint operation equation (2.24), there are subtle differences. As with the other component types, the inlet temperature is a known value, and the specific heat is essentially a fixed (explicit) value for a given step.

The first major difference is that for loop setpoint components, the heat transfer rate is applied by the loop solver as a portion of the loop demand. This is prescribed based on current control strategies and constraining conditions (limiting temperatures). Dispatching the load is a flexible aspect of the simulation model in that the total loop demand can be distributed to multiple components using one of three supervisory control schemes:

Uniform: All components are loaded equally among all available equipment.

Optimal: The components are loaded in an attempt to meet a part load that is defined as optimal in user input.

Sequential: Components are loaded incrementally, with the first component loaded fully before any additional components are loaded.

This leaves two variables to be solved from equation (2.25): $\dot{m}$ and T_{out}. Once the flow has been locked by the solver, the mass flow rate is fixed and so the outlet temperature is simply calculated as the outlet boundary. For cases when the flow is unlocked, the equation is closed by constraining either one of these values based on operation. For components which have no mechanism for providing flow control (constant flow), a design flow rate is always requested from the loop flow resolver. In this case the outlet temperature is simply the result of solving the equation. The other option is for components which have the ability to control their own flow rate. In these cases, the components must again have a target outlet setpoint, such that the mass flow rate can be calculated from the above equation. Note there is a similarity and a distinction between two individual situations, in which both cases the entering temperature and outlet temperature (setpoint) are specified by the solver and control strategies:

Component Setpoint: These components are utilized to meet a local component outlet setpoint.

Mass Flow Rate: Component setpoint controlled components can have a design flow rate that is constant.

Heat Transfer Rate: The heat transfer rate is solved in order to meet the local outlet setpoint condition.

Loop Setpoint Component with Variable Flow: These components are used to meet loop-level demand, but require a setpoint on the component outlet.

Mass Flow Rate: The mass flow rate for this component is variable.

Heat Transfer Rate: The heat transfer rate is imposed by the control strategies as part of the load dispatch calculations, which may include distribution to multiple components.

2.3.5 Flow-wise Simulation Mechanics

This section describes the actual simulation mechanics for a given loop-side. This includes the following discussions:

- The flow-wise simulation approach (Section 2.3.5.1)
- Handling concurrent control strategies (Section 2.3.5.2)

The goal of this section is to close the discussion of how a single loop-side is simulated while attempting to maintain control.

2.3.5.1 Flow-wise simulation approach

The approach for component simulation and the loop solver have been described, including the mechanisms for providing flow resolution (Section 2.3.2), and handling loop demand (Section 2.3.4). The method used in the proposed ICE-B model to actually connect the system of components to provide a full solution is described here. The existing simulation model did not perform a pure flow-wise solution, instead relying on simulating certain component types before other component types which caused issues with out-of-date boundary conditions being employed on component models.

In the proposed model, components are always simulated in a flow-wise fashion. From a starting point on the loop, an inlet boundary condition is prescribed, and this is provided to the first encountered component. This component will use this well-defined inlet boundary and other conditions to perform calculations that result in, among other things, an updated outlet state point. This is then passed to the downstream component, which is simulated next. This continues until the entire loop has been simulated. This completes one iteration of the loop solution. This successive substitution continues until the loop is converged, which is described in more detail in section 2.3.6.

The flow-wise mechanics do not only apply to single flow paths, but also to parallel paths, which are encountered in many loops. While the flow resolution of these parallel networks was described in section 2.3.2.5, the loop demand and heat transfer implications of a flow-wise solution was not. During the simulation of components on a loop-side, a splitter (see Figure 2.3) has the responsibility of providing meaningful boundary conditions to the inlet components on each of the parallel paths. Splitters are modeled as ideal objects, so the temperature passed to components downstream of the splitter is exactly the splitter inlet temperature. The flow rate distribution is specified by the flow resolution engine, however, the splitter will also pass metadata such as maximum available flow to ensure that components abide by restricted flow conditions and pump capabilities.

Like splitters, mixers are ideal objects, but must perform calculations to provide a correct inlet boundary condition for downstream equipment. The mixer outlet mass flow rate is a simple solution to the continuity equation:

$$\dot{m}_{mixer,outlet} = \sum \dot{m}_{mixer,inlet} \tag{2.26}$$

The temperature is also a simple solution, this time to the energy equation, which reduces to a flow-weighted average temperature:

$$T_{mixer,outlet} = \frac{\sum \left(\dot{m}_{mixer,inlet} T_{mixer,inlet}\right)}{\sum \dot{m}_{mixer,inlet}} \tag{2.27}$$

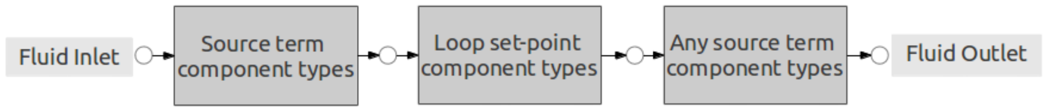
By simulating the entire loop flow-wise, both components and these connectors, the result is a fully updated loop, or one step in the simulation algorithm.

2.3.5.2 Concurrent Control Strategies

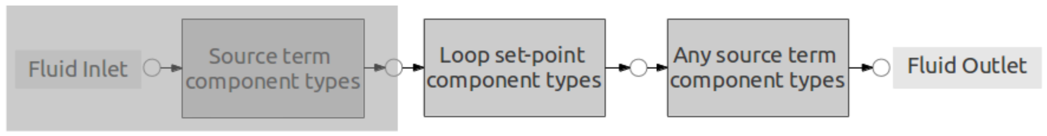
Components are simulated flow-wise as described in the previous section. Each flow-wise sweep is divided into a maximum of three possible phases, based on the diversity of control strategies in place on the loop-side. The use of multiple concurrent control strategies (loop-setpoint, component-setpoint) on the same loop-side is a new feature in the proposed ICE-B model.

Series Path Consider a loop-side that contains a set of uncontrolled/source term components, followed downstream by a set of loop-side components, followed by another set of uncontrolled/source term components. This is shown in Figure 2.12a. The simulation moves flow-wise along this series path, simulating component models as they are encountered.

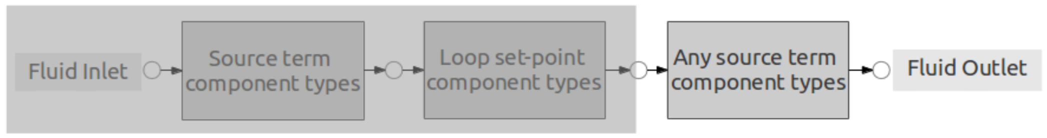
The flow-wise simulation begins at the loop-side inlet, simulating any and all source-term type control components (uncontrolled, component-setpoint) and continuing until a loop-setpoint component is found. At this point, the components which have been simulated are highlighted in Figure 2.12b. The components simulated will have affected the loop demand source term, and this is monitored by the solution algorithm, to eventually achieve an adjusted source term. Once a loop setpoint component is encountered, the entire loop demand (sum of the original loop demand plus any source additions) can be dispatched to all available loop-setpoint


Fluid Inlet
Source term component types
Loop set-point component types
Any source term component types
Fluid Outlet

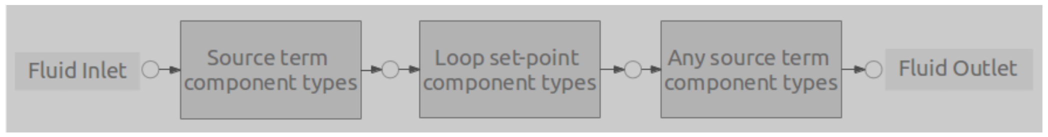
(a) Series loop with source components, loop-level controlled components, and additional source components


Fluid Inlet
Source term component types
Loop set-point component types
Any source term component types
Fluid Outlet

(b) Highlighted region shows components which have been simulated in the first step, which includes any upstream source term components


Fluid Inlet
Source term component types
Loop set-point component types
Any source term component types
Fluid Outlet

(c) Highlighted region shows components which have been simulated in the second step, which includes all loop-level controlled components


Fluid Inlet
Source term component types
Loop set-point component types
Any source term component types
Fluid Outlet

(d) Highlighted region shows components which have been simulated in the final step, which includes any downstream source term components

Figure 2.12: Demonstration of the three steps in a loop-side with multiple concurrent control strategies

controlled components. With the load dispatched, the simulation moves flow-wise through the loop-setpoint controlled components. When complete, the components which have been simulated are highlighted in Figure 2.12c. If there is sufficient capacity, the loop is able to meet the outlet setpoint. However, the simulation continues flow-wise, and if further source-type components are encountered downstream of these controlled components, the loop may no longer meet setpoint. The solver is fully capable of handling this situation because there may be occasions where uncontrolled components such as economizers may be optimally placed at various locations around the loop. At this point, the highlighted components in Figure 2.12d represent that the entire loop-side has been simulated.

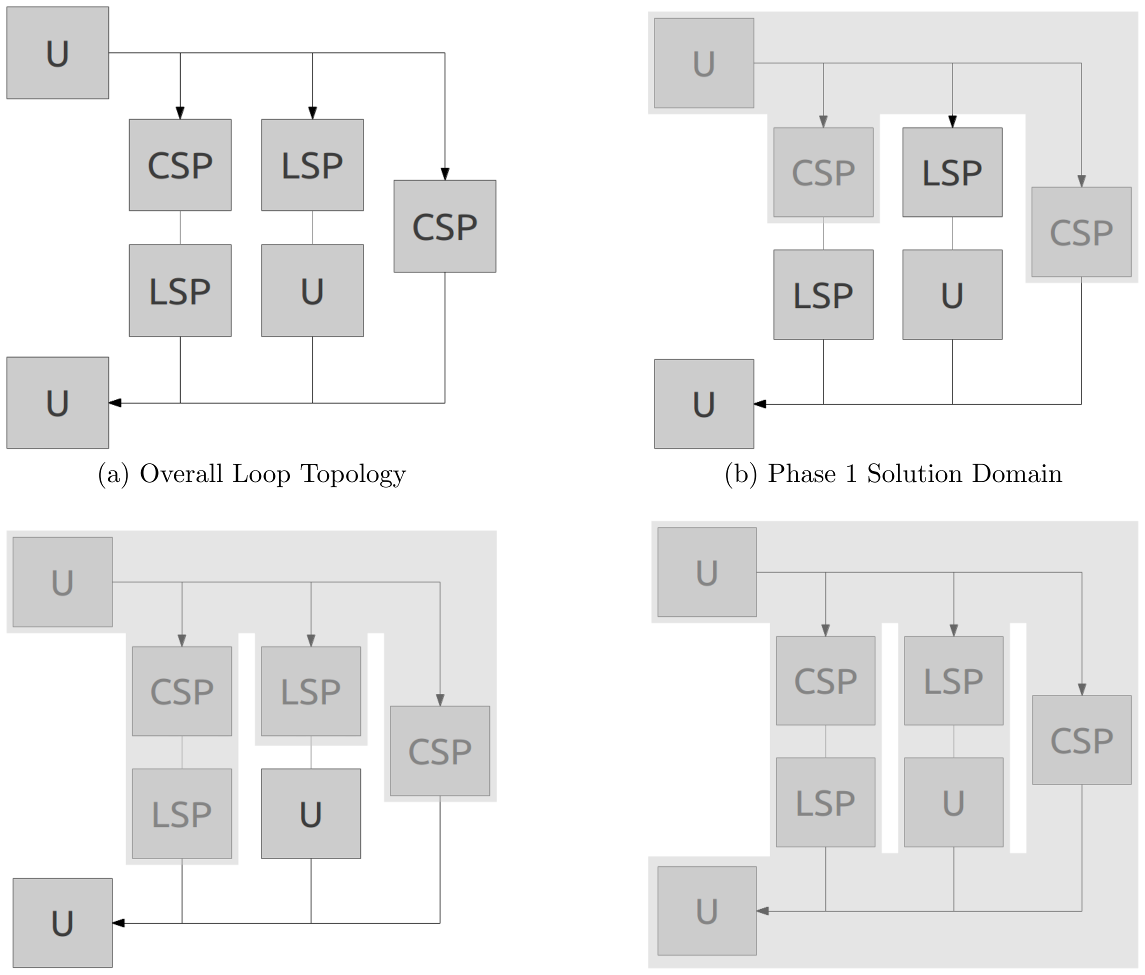
Parallel Paths The previous discussion related to the three phases of component simulation in an attempt to provide a controlled solution by the loop solver, even on loops with diverse, concurrent control strategies operating. This was described within the context of a single flow path. The same approach is utilized in situations with parallel flow paths. The approach taken to accommodate this is to force each phase of the solution to propagate through all available paths before moving to the next phase. This is an important feature of the implicit solution mechanics and allows the solver to provide a controlled condition in all possible cases. This feature can be demonstrated using a parallel loop as shown in Figure 2.13, with the following shorthand:

U: Uncontrolled component

CSP: Component SetPoint control component

LSP: Loop SetPoint control component

The basic loop is first shown in Figure 2.13a. The loop contains a variety of component types. This may seem like an exotic loop configuration, but consider that uncontrolled equipment may consist of economizing or free cooling heat exchangers, component setpoint components may represent chillers that provide specific feed temperatures to thermal storage tanks, and loop setpoint components may be the components used to provide an overall chilled water supply temperature. The loop shown may appear with an unusual arrangement of components, but is designed to exercise the solver and demonstrate the three phase solution clearly.



(a) Overall Loop Topology

(b) Phase 1 Solution Domain

(c) Phase 2 Solution Domain

(d) Phase 3 Solution Domain—Done

Figure 2.13: Three phase solution domains

Before any components are simulated, the inlet condition of the loop along with the loop setpoint temperature are used to evaluate an initial loop demand, the first term in equation (2.22). This baseline demand is then adjusted via the source term as other components are encountered, whether in series or in parallel.

The first phase of the solution involves simulating all source term type components on the loop in a flow-wise fashion. The components simulated during this phase are highlighted in Figure 2.13b. The inlet leg of the system contains an uncontrolled component, which is simulated first. This component may add or remove heat from the loop. The solver monitors this and adjusts the remaining loop demand accordingly. The solution moves through the flow splitter and begins simulating the parallel legs. The first leg, first component is a component setpoint operation component, thus it is also simulated as a source term component. The solver continues to adjust the remaining loop demand after this component is simulated. The next component is a loop setpoint control component, so at this point the solver stops solving on this flow path, instead moving to the next parallel leg. On this leg, the first component is loop setpoint, so again the solution stops and moves to the next parallel leg. On the third leg, the only component is component setpoint control, so the solver simulates this component and continues to the mixer. However, the solution does not propagate *through* the mixer until all parallel legs are completed to ensure the mixer has correctly updated boundary conditions available.

At this point the first phase is complete, so the solver has a well-defined prediction of the loop demand using the baseline and adjustments via the source term. The second phase then proceeds to simulate loop setpoint control components. The simulation begins back at the first parallel leg where it left off. As the first loop setpoint control component is encountered, load is dispatched to *all* available loop setpoint control components in a uniform, sequential, or optimal fashion (see section 2.3.4.2). The simulation then simulates all loop setpoint components in a path-by-path flow-wise fashion until the highlighted state of Figure 2.13c is attained. Note that the solution stopped in the second parallel leg when an uncontrolled component was encountered.

At this point, the second phase is complete. If there were no components remaining on the loop, and assuming that there was sufficient capacity and a proper controls implementation, the solver would have utilized the controlled components to meet a loop setpoint condition. If, however, there are remaining non-loop setpoint components on the loop, the solver enters phase 3. Phase 3 involves the solution of all remaining components on the loop. For this loop, one uncontrolled component required simulation in the second parallel leg, followed by the mixer, followed by another uncontrolled component on the outlet leg. If these components affect the loop by adding or rejecting heat, the system may not meet setpoint, but the solver will have successfully simulated the loop-side.

At this point, the loop has been simulated as shown in Figure 2.13d. Note this design has an inherent restriction of the placement of component types. Component setpoint or uncontrolled components may not separate loop setpoint components on a single flow path. This is because the effect of these other components are not well-defined and the solver cannot predict their effect on the loop. Thus if any loop setpoint components were encountered during phase 3, the solver would issue an error, alerting the user of an invalid topology.

2.3.6 Supplemental Feature Discussion

The methodology employed in the proposed ICE-B model to simulate a single loop-side, for both flow and heat transfer solutions, has been established. This section now discusses remaining pieces to the entire simulation model that did not appropriately fit in the discussion elsewhere, and/or were not developed predominantly by this author, but are still relevant to closing the discussion, and providing full content for the forthcoming model demonstration (Section 2.4).

2.3.6.1 Component/Loop Sizing

Many components and systems within the EnergyPlus simulation program can be autosized. These include zone equipment sizing, air loop component sizing, and plant sizing. In this way, a building may be simulated in any climate, and the required system components can be sized to match the load requirements of the zone. This allows for easy parametric studies, especially those that involve climatic variation. As an example of enhanced sizing, the building energy management system (EMS) in EnergyPlus allows systems to be sized to the appropriate climate, and then user-defined programs can be utilized to increment component sizing to meaningful values (tonnage increments available for component types, for example).

As a part of the ICE-B model development, the sizing routines were updated, however not by this author. As such they are not described in great detail, but are worthy of a brief overview as they are a core aspect of the simulation environment. In the existing model, component sizing took place in a single solution step: peak plant demand conditions were tallied once in order to size required plant loop flow rates, while desired design delta temperatures and related information were used to size component capacities. The new plant simulation algorithms provide an improved integration between air loops and multiple plant loops, and as such the sizing is also improved. The sizing algorithms utilize an iterative approach to provide a solution to the sizing problem. The plant loops are simulated in the predetermined calling order, and at each update, the demanding components will request a flow rate from the plant, which will in turn cause the plant components to size to a certain capacity. If this component is then connected to other nested loops, the updated capacity will affect the sizing of the nested loops. This iterative approach ensures that the entire system of loops is able to size to a meaningful solution, which provides a much better estimate over the straight-through approach.

2.3.6.2 Common Pipes

Common pipe simulation is a major aspect of the thermal plant model that is utilized in many real system applications. This was another aspect of the upgrades to the simulation model that was not performed directly by this author, but is coupled with the plant simulation. The common pipe provides a way for demand (secondary) and supply (primary) systems to operate at different flow rates as a measure to reduce excessive energy use.

This is modeled by solving a system of equations. In the controlled case, there are six equations and six unknowns which govern the common pipe simulation, consisting of mass balances at each node on the system, an equation governing the loop capacitance tank, which is connected directly to a node, and also a setpoint node which has a trivial update equation. This system is solved iteratively at each plant iteration to achieve local convergence and also convergence within the plant system.

2.3.6.3 Loop Capacitance

In previous versions of the plant simulation algorithm, the purpose of loop capacitance was two-fold:

1. Effectively represent the physical capacitance of the loop
2. Provide stability to the loop simulation algorithms.

Loop capacitance can be considered as an effective mass of the system. The plant simulation solver is a quasi-steady solution. To provide a massive, transport delay effect, the loop capacitance was historically modeled as a single well-mixed tank that existed at one interface between the demand and supply loop-sides. This has been improved in two ways in the proposed ICE-B model:

1. The tank can be split into two separate tanks, one placed at the inlet of each loop-side. This was performed to spread out the capacitance beyond one single

point in the system.

2. The tanks are now the sink for the inclusion of pump heat in the simulation model. In the existing model, the pump heat was added directly at the pumps, providing an instantaneous temperature increase from pump inlet to pump outlet. In reality, the energy added by the pumps is not instantaneous, but rather is added as kinetic energy to the fluid. This energy is released as friction and diffusion around the system. In the proposed ICE-B model, the pump heat is now lagged behind the simulation by one system time step and placed as a heat gain on the loop-side tank.

In addition to the tank models, a transport delay model based on actual pipes which can be distributed around the loop in more detailed fashion was added to the EnergyPlus library. This discussion is found in chapter 3.

2.3.6.4 Loop Convergence

The simulation model uses iteration to solve the system of state points in an implicit, successive substitution manner by simulating component models until convergence is obtained. Iteration and convergence control are implemented by checking for deviations between the outlet of one loop-side to the inlet of the connected loop-side, and by monitoring a number of state points around the loop. The thermodynamic state is checked, and in addition, the maximum available flow is continuously monitored to check for disagreements between the loop and components. In the proposed ICE-B model, this has been enhanced to leverage the new mass flow request system. When a component continuously requests more flow than available and adjusts its flow request between iterations, the loop may continue iterating unnecessarily, so monitoring this is important, and checks are employed to break the continuous loops.

2.3.6.5 Coupled Loops

Most of the discussion here has been for a single loop-side, as this solver is designed flexibly to accommodate any single loop-side. Some discussion went further to include full loops (two loop-sides). In a real system, and in the simulation model, multiple loops may be used and coupled together. Consider a simple chilled water system including a condensing loop with a cooling tower as shown in Figure 2.14.

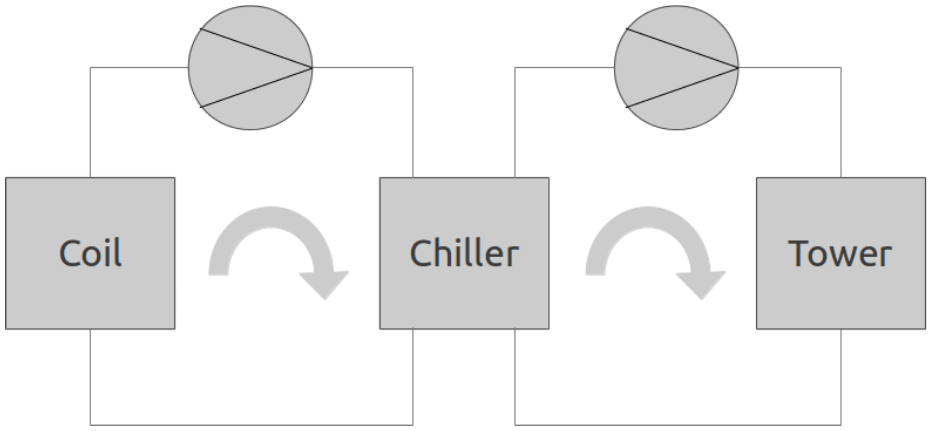


Figure 2.14: Coupled loops in a condensing chilled water system

In the existing EnergyPlus model, the nested nature of coupled loops is not recognized. Instead, loops are simulated in order of type:

1. Simulate all plant demand sides
2. Simulate all plant supply sides
3. Simulate all condenser demand sides
4. Simulate all condenser supply sides

The proposed ICE-B simulation respects the interdependence of these loops when determining a proper simulation order of all the loops, however components that couple loops together are treated the same as any other single loop component by the solver. The coupling is performed by the component model itself using the *SetComponentFlowRate* interface as described in section 2.3.2.3. The component is able to provide flow requests and heat transfer impacts on both of the connected loops so that as each loop is simulated it will have up-to-date requests. Ultimately through iteration these flow requests converge, thus converging the entire system of loops.

2.3.7 Summary

The model methodology includes many different aspects, which are implemented to simulate a diverse set of possible system topologies and configurations. The methodology discussion began by introducing the idea of a component model, including model responsibilities and the type of governing equations to be solved by these components. The discussion then shifted to the flow network solution, and how this is performed using a logical iterative approach of flow-request/flow-resolution rather than requiring a full pressure network solution. Pumps are implemented in a different manner than in pressure-based modeling environments, and the discussion of these models included the contrast between flow rate and rotation speed, constant and variable flow, and pump banks. A discussion of how a pressure simulation was layered over the base solver was provided. The final discussion included the way components and the solver are actually coupled, followed by big-picture loop simulation mechanics and topics which did not fit elsewhere, but are important to providing an overall understanding of the proposed simulation model.

This methodology was implemented as an overhaul of the previous central plant simulation engine in EnergyPlus and was released to the public domain in 2011 by the United States Department of Energy (United States Department of Energy, 2012). Before this upgrade, there was a high level of burden put on developers in maintaining several pieces of the plant simulation engine. The most notable and frequent bugs related to enforcing continuity in the system. This was due to an inherent flaw that component models were allowed to *specify* their flow rate at any time. This was corrected in the new model design by providing the interface between component models and the solver which allowed proper flow request/resolution logic to be enforced, and ensure that the system achieves continuity under all conditions. After the release of the new engine (and brief shakedown period), the number of issues lodged by users diminished to nearly zero. This has reduced the maintenance burden and allowed developers to focus on new features and enhancements rather than struggle to maintain poor code.

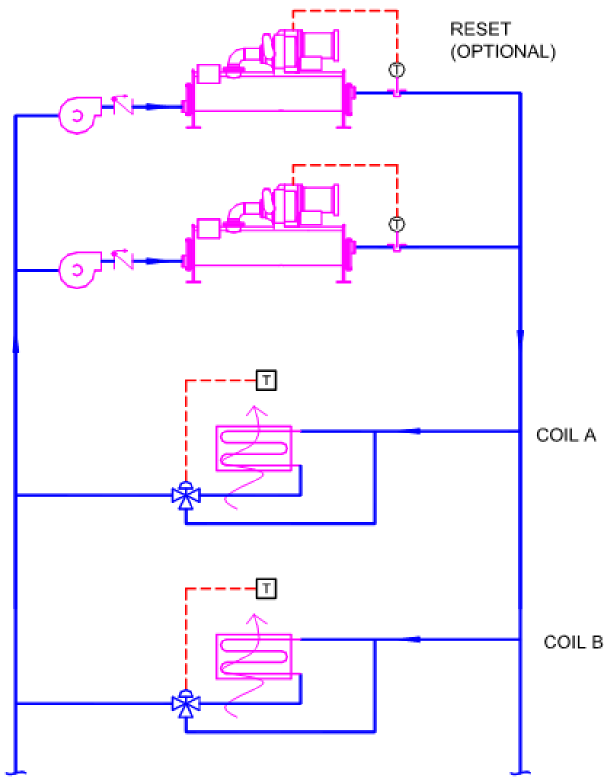
2.4 Model Evaluation

This section demonstrates the ability of the new thermal plant simulation model in EnergyPlus to simulate physical systems. Each case consists of a description of the physical system, followed by model abstraction, a demonstration and analysis of results, and conclusions. The process of abstracting each system into model form includes:

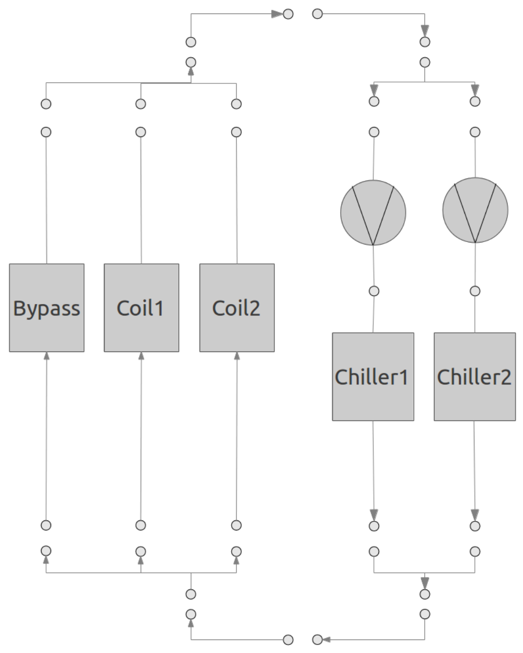
- System Topology: the placement of components and flow paths on the simulation loop and how it is similar and different to the physical system
- Components: the selection of component models to match the physical system, and what components are missing/implied in the simulation environment
- Controls: how physical system control strategies are employed in the simulation environment, and consequences of using specific strategies

2.4.1 Case A

The first central plant system to be analyzed is shown in Figure 2.15a. The schematic includes two chilled water coils which comprise the demand side and two chillers which comprise the supply side. Each chiller has a dedicated constant speed pump and valve. The chillers are controlled using a fluid temperature measurement at their outlet. The coils each have a bypass leg to meet a zone air setpoint. While there are two coils shown, the diagram implies that others may exist, but this discussion need not include these to demonstrate the abstraction process.



(a) Physical system schematic, courtesy of Steve Taylor (used with permission)



(b) Simulation Loop Schematic

Figure 2.15: A constant speed pumping system with multiple chillers and multiple coils; dedicated pumps

2.4.1.1 Topology

The first step in abstracting this physical system into model form is determining how the system will actually "look" in EnergyPlus. In EnergyPlus input specification, there is a distinction between demand and supply sides of the fluid loop (although the underlying solver is generalized for any loop-side). The input specification for this case is included in full in Appendix A. In this physical system, the demand side consists of two coils in parallel, with no common bypass, although each coil has a local bypass leg to control flow. Since there is at least one bypass path on the demand side, the model topology is required to include a common bypass. The supply side consists of two parallel legs that each have a pump, flow control valve, and a chiller. This fits well with the simulation loop topology, which was described in section 2.1.3. Piping segments are added in specific points in the loop in order to "connect" these components together. There are also mixers and splitters included to connect the parallel sets to the rest of the loop. The resulting simulation loop topology is shown in Figure 2.15b.

2.4.1.2 Components

The component models used in the simulation will determine the energy transfers and efficiency of the whole system, so proper selection is important. For this system, there are only four simulation components: the coils, the pumps, the chillers, and connector components (pipes).

- **Coils**: The physical coils are chilled water to air coils. While the coil simulation models include the ability to request a varying amount of flow in order to exactly meet the zone setpoint, approximating the use of the bypass valve in the physical system, the input is simplified and uses a load profile object. The load profile objects are specified in the input listing at lines 303 and 345. The load profile input consists of a scheduled flow request, heat transfer rate addition to the loop, and inlet and outlet node names. The simulation management uses these names along with other topology input specification to determine the flow-wise order and placement of components along each loop-side.

- **Pumps**: The pumps are constant speed in the physical system, which does not indicate a constant flow rate, but instead a constant pump rotational speed. The constant speed pump model in EnergyPlus is closer in nature to a constant flow rate pump model (see section 2.3.3). The pump will try to run at a design condition unless there is a flow restriction around the loop that won't allow it to run at full design flow. The pumps are specified in the input listing at lines 165 and 207, and include design flow, pressure head (for calculating energy use only), efficiencies, and inlet and outlet node names.
- **Chillers**: The chillers can modeled using, for example, curve fit or parameter estimation forms. Certain model forms fit certain chiller types better. The model selected for the current work consists of a constant COP, which minimizes the overhead in creating inputs for the system model. The system simulation relies on a predefined flow direction in the loop, so the check valves are not required. The chillers are specified in the input listing at lines 176 and 218, and include design flow, COP, capacity, and inlet and outlet node names.
- **Pipe Connectors**: The pipes are specified in multiple places around the loop, including the bypass components, and so there are multiple locations in the input file where pipes are specified. For these simple components, the inputs include only a name, and inlet and outlet nodes.

2.4.1.3 Controls

This physical system is controlled via four temperature measurements: fluid temperature at the outlet of each chiller and a zone air temperature corresponding to each coil. For the coils, the zone air temperature controls a three-way valve which allows some fluid to bypass the coil to attempt to meet the zone air setpoint. A single bypass leg in the model abstraction handles all bypass flow. For the chillers, the fluid outlet temperature itself is a setpoint which is used to control chiller compressor cycling. The simulation model can utilize similar controls. The valves for the coils are implicit in the coil model formulation. The coils request fluid flow from the plant at a variable rate without explicitly changing valve settings. The coil controller attempts to optimize this flow request to exactly meet the zone demand (and therefore allow the zone to meet the setpoint temperature). For the supply side, the flow valves are again implicit, due to the predefined flow direction in the model. The chiller control can be set to an outlet temperature setpoint. In the simulation model, this is termed "Component setpoint control." The chillers can also be staged to operate under various staging strategies. These strategies include a sequential operation where the chillers are brought on one-by-one to meet demand. Uniform loading is also possible which brings all equipment on equally to meet demand. Optimal loading is also implemented to attempt to load components to an optimum part-load ratio when possible. (See section 2.3.4 for further information on controls.)

2.4.1.4 Demonstration 1

The physical system was modeled in EnergyPlus using load profile objects to mimic the coil flow requests and heat demands. This simplifies the inputs such that a full zone and air-system implementation are not required. The load profiles use scheduled flow requests and heat demands. If the loop cannot meet the demand of the load profiles under certain conditions, it will be equivalent to not being able to meet a zone setpoint. The chillers were modeled as electric air-cooled chillers to alleviate the need to add a secondary condenser loop and tower/ground heat exchanger to the system. A dedicated pump was placed on each chiller leg and modeled as a constant speed pump.

The scenario was set up so that the load profiles would add demand to the loop, at a varying rate through a given simulation day, with flow rates varying along with the demand. The chillers were both controlled to an outlet temperature setpoint, and set to be staged sequentially. Only the equipment necessary to meet the demand should run, with the rest of the equipment remaining dormant.

The outputs from EnergyPlus include:

1. Load Profile Demand and Actual Heat Transfer Rates
2. Load Profile Requested and Actual Mass Flow Rates
3. Chiller Mass Flow Rates
4. Chiller Evaporator Fluid Mass Flow Rates
5. Chiller Outlet Temperatures and Mixed Chilled Water Supply Temperature

The outputs from each of these classes are plotted in separate figures in Figure 2.16.

The first plot in Figure 2.16 shows the load profile (scheduled) demand and (actual) heat transfer rates. The demand and heat transfer rates are equal throughout the simulation for each load profile (though load profile #1 has a higher (more negative) demand in the morning, to mimic load diversity).

The second plot shows that the load profile (scheduled) flow requests equal the actual operating mass flow rate.

The third plot shows the chiller fluid mass flow rates. Note that even when the demand is low in the morning, and the first chiller alone should meet the demand, both chillers ramp to full flow capacity. Also note the difference between the sum of the load profile flow rates and the sum of the chiller flow rates. The difference indicates that there is bypass flow on the demand side. The reason for this excess flow will be explained and corrected in demonstration 2.

The fourth plot shows the chiller fluid heat transfer rates. Note that instead of the first chiller ramping up to take the loop demand and the second staying off, both

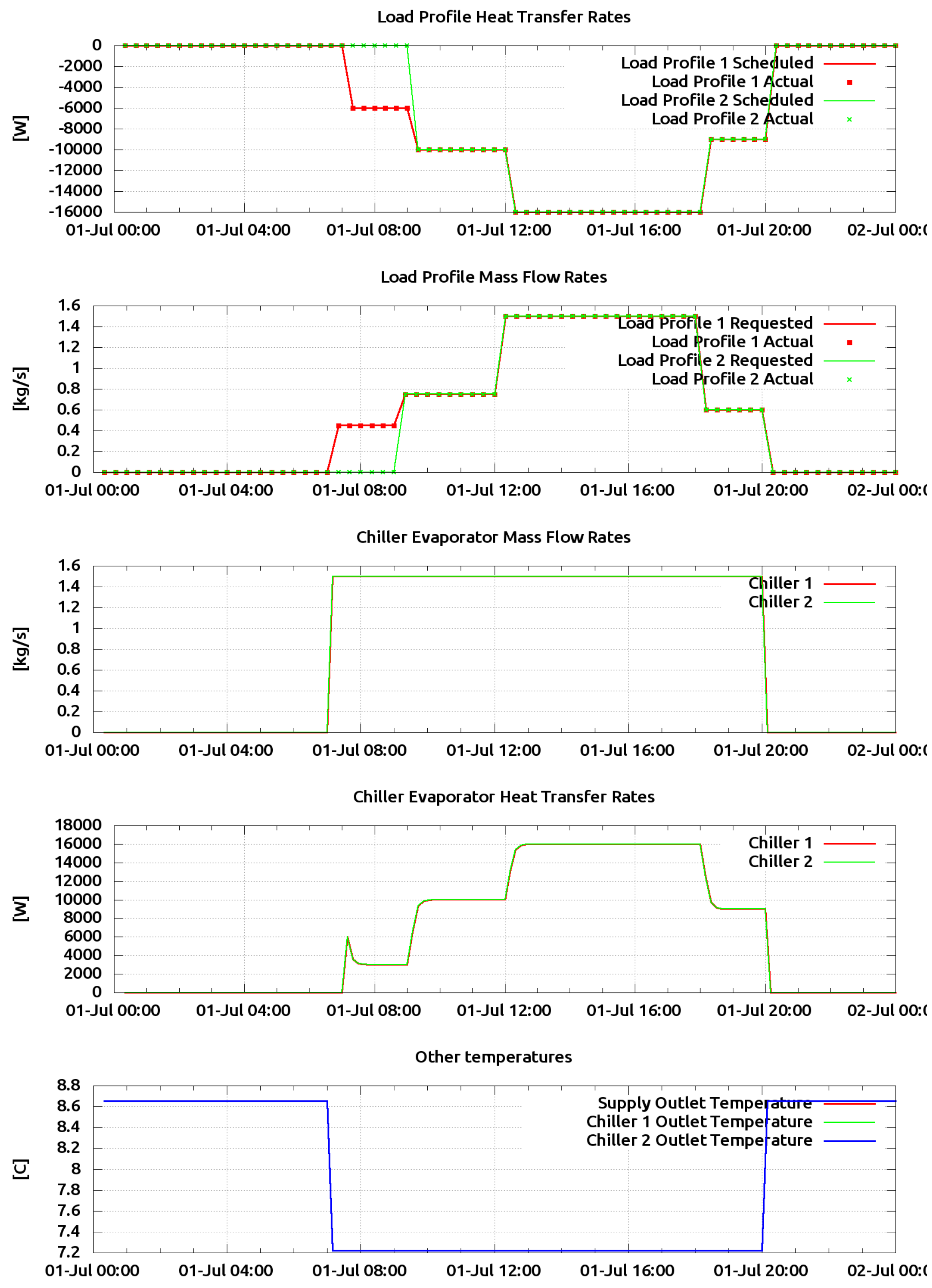


Figure 2.16: Results: Operating on a component setpoint scheme

chillers reject heat from the loop at equivalent rates, which vary throughout the day. The heat transfer rate does not exactly match the loop demand added by the load profiles, but follows a similar trend. The imbalance is due to two things:

- Pump heat gain, which causes a higher heat transfer rate, though this is a small fraction of the loop demand (in this case)
- Loop capacitance, which causes the dampening effect seen as the loop conditions vary. This loop capacitance, along with pump heat, also causes the initial pick-up load which must be rejected from the loop to bring the loop to operating temperature.

The fifth plot shows temperatures at the outlet of each chiller and the mixed chilled water supply temperature. Note that anytime the loop is active, the temperature at each chiller outlet is exactly the setpoint temperature, which when mixed will be the same temperature, so that all plot results overlay each other.

2.4.1.5 Analysis 1

The reason for the unexpected excess flow is the localization of the chiller controls and pump model mechanics. The chiller component-setpoint controls are simple:

> If the loop is on already, turn on chiller and attempt to meet outlet setpoint.

When either one of the load profiles turn on, flow is requested and the loop will turn on. With the loop on, the chillers will both attempt to meet their outlet temperature setpoint. Since flow is available, each chiller's dedicated pump will turn on to meet this flow request. Since the pumps are constant speed pumps, they both ramp up to full speed. This induces a "full-flow" condition on the supply side of the loop. When the flow comes back to the demand side, the bypass accepts the excess flow, allowing each load profile to only take their requested flow. This brings the entire loop to full flow, and both chillers are running to meet their outlet setpoint.

As mentioned, the reason for this running condition is the localization of chiller controls and pump model mechanics. Firstly, the pumps are dedicated constant speed pumps. When any flow is requested of them, they offer to run. Since the loop is not restricted (a valve physically closed, for example) both pumps can run at full capacity. With the loop turned on, and the chiller controls localized to the chillers themselves, the only option the chillers have is to turn on and try to meet their local outlet setpoint. The setpoints may be scheduled in EnergyPlus, but optimal setpoint reset that would achieve desired chiller sequencing must be abstracted in the EnergyPlus model as a loop-setpoint. The loop-setpoint acts as an ideal temperature reset control for the system shown in Figure 2.15a.

2.4.1.6 Demonstration 2

To improve this model, a supply side loop control scheme is utilized. Instead of each chiller responding individually to try to meet their specific outlet node setpoint temperature, a mixed supply water temperature setpoint is employed in contrast to the localized chiller outlet setpoints in the initial version. This differs in that a total loop demand is then distributed to the individual machines in order to properly distribute the load. The chillers will not turn on solely to satisfy their local outlet setpoint, but only run when the demand requires them to run. The results of changing from a component setpoint control to a loop setpoint control is shown in Figure 2.17.

In Figure 2.17, the first plot, showing load profile demand and heat transfer rates, is exactly the same.

The second plot shows that now the load profile mass flow rate requests are still equal to the actual mass flow rate.

The third plot shows the chiller fluid mass flow rates. In the morning hours, when

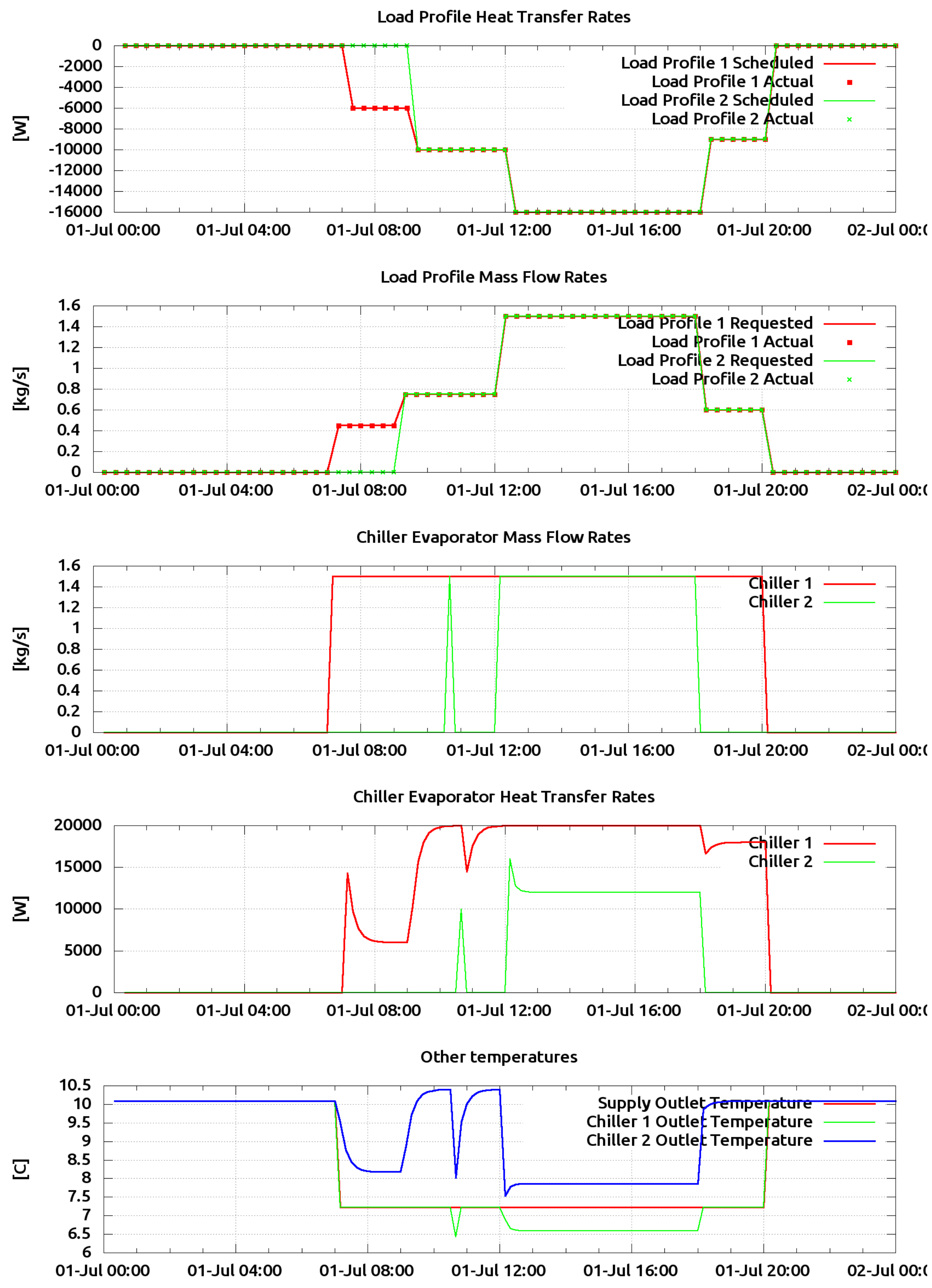


Figure 2.17: Results: Operating on a loop setpoint scheme

the demand is capable of being met with a single chiller, only one chiller runs. During the afternoon when there is a major heat addition to the loop by the load profile, both chillers must run. Note that whenever either chiller is running, it runs at full design flow rate. This is due to the constant speed pumps assigned to each chiller. Even if the chiller would like to run at a part load flow rate, the pumps will always try to run at design. Note also in this plot that there is a spike at about the 10:45 mark. This is an interesting artifact and due to the loop capacitance. Initially, some of the heat being added to the loop by the pumps and the load profile is stored by the loop capacitance model. At this point in the simulation, with a transient condition of the second load profile turning on, enough heat is added back into the loop to require the second chiller to turn on. Once this initial pick-up load is rejected and the loop temperature gets back under control, the first chiller can then meet the loop demand and the second turns off until again needed in the afternoon.

The fourth plot shows the chiller evaporator heat transfer rates. The chillers are now not being equally loaded to meet their outlet setpoints. Instead they are being loaded by a dispatch algorithm to meet the total loop demand. Chiller 1 is loaded first with a brief pick-up load followed by first-order responses to changes in loop demand. These curved responses have little to do with the actual chiller response, and are instead artifacts of the loop capacitance model (tank). The second chiller is shown to respond when the loop demand is high enough to necessitate it, but is never loaded to full capacity.

The fifth plot shows temperatures at each chiller outlet and the mixed chilled water supply temperature. Now that the chillers are controlling to a loop setpoint, the individual chiller outlet temperatures have been optimally "reset" at each timestep. Chiller 1 often has a temperature below the chilled water setpoint temperature, to accommodate the mixing that occurs with the fluid passing through chiller 2, which is often just a floating system temperature. The loop is able to control to the mixed (chilled water supply, for example) setpoint temperature through the entire day when a load is present.

2.4.1.7 Analysis 2

With constant speed pumps dedicated to each chiller, when either chiller comes on, the full flow of the pump will be present. This is independent of the flow being requested by the demand side. When this extra flow passes through the demand side, the flow resolution algorithm distribute the excess flow to the bypass leg. This allows the supply side components to run at a fixed flow rate, while allowing the demand side components to run at a variable flow rate; all while still maintaining continuity around the loop. This is similar to how primary-secondary systems are commonly set up, though in those cases, each side of the loop will have a pumping setup.

2.4.1.8 Demonstration 3

At this point, the loop is behaving as expected based on the system schematic, and intuition about the system. However, further advances can be explored to reduce the energy use of this system. One obvious measure would be the addition of variable speed pumping. This would reduce the excess flow around the loop and only run at a required part load ratio. Variable speed pumping in the simulation is interpreted as variable flow. The pump is capable of responding to flow requests around the loop, rather than trying to always run at design flow. The results of using variable speed pumping are shown in Figure 2.18.

In Figure 2.18, the first and second plots are the same as the last versions. The load profile heat transfer rates and mass flow rates are also exactly equal to the requested variables.

The third plot shows the chiller fluid mass flow rates. The chillers are brought on sequentially to meet the demand, and operate at the flow rate required to meet the

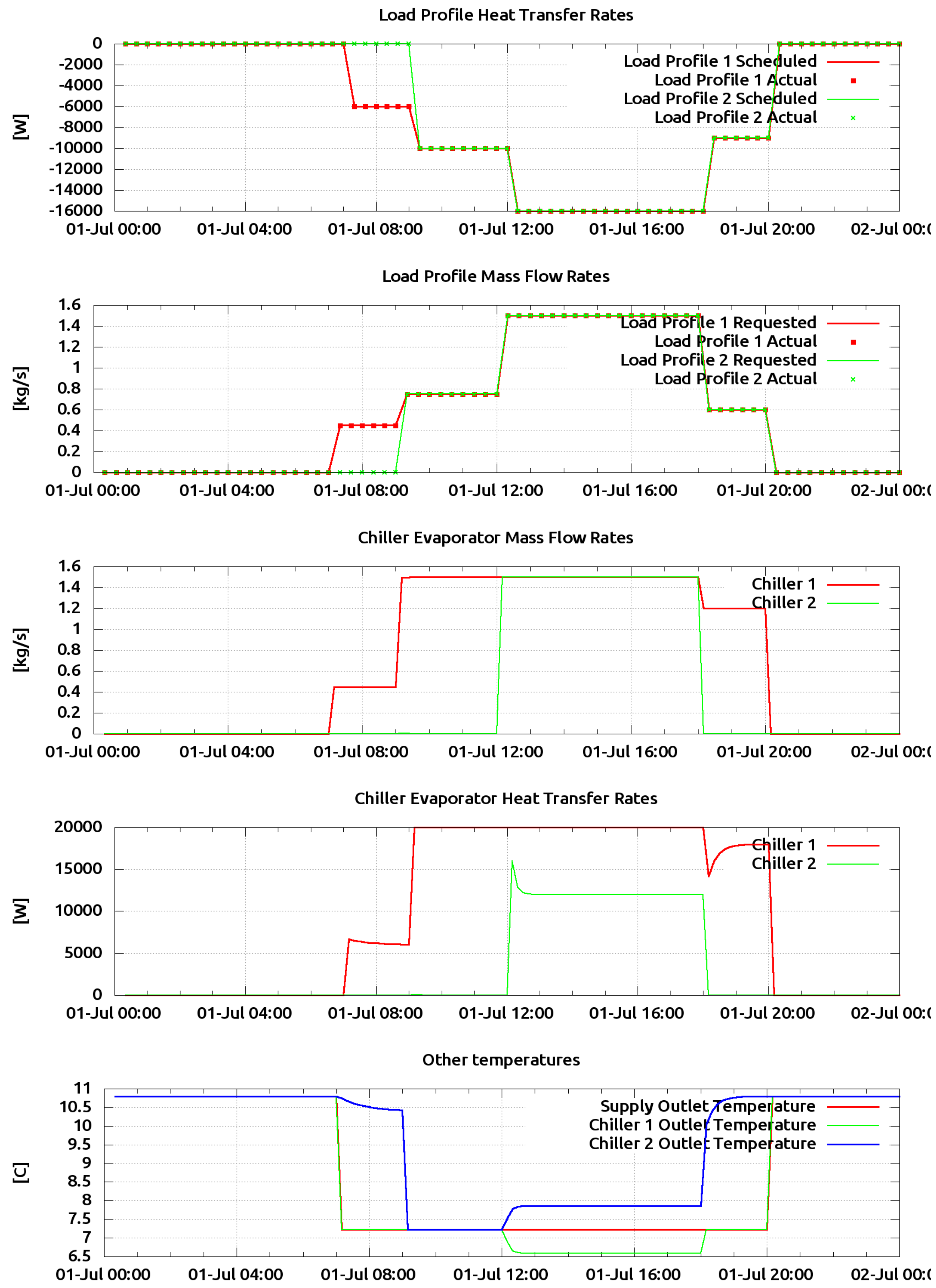


Figure 2.18: Results: Operating on a loop setpoint scheme with variable speed pumps

demand. In fact, the chillers will only run at the flow requested by the load profile. This is equivalent to the chillers meeting both the loads and flow requests of the coils in an actual system. Previously, with a constant speed pump system, any additional load caused the pumps to come on to full design capacity, which resulted in a large heat addition to the fluid, which caused spikes. With this variable speed system, the pumps are ramped up smoothly which avoids the spiking problem, although small pick-up loads are still encountered and expected.

The fourth plot shows the chiller heat transfer rates, which are similar to the previous version, only smoothed out due to the variable speed pumping.

The fifth plot shows temperatures around the loop. Note that during the first portion of the morning, the second chiller is not running, but still reports a decreasing fluid temperature as the entire loop is cooled. In the afternoon, chiller 1 also shows some subcooling below the setpoint in order to properly mix with the second chiller and ultimately hit a mixed chilled water supply temperature setpoint.

2.4.1.9 Analysis 3

The variable speed pumping operation reduced the excess flow conditions around the loop which would conserve a significant amount of energy in a real system. The use of variable speed pumping also smoothed system transients in the simulation model, though analogous improvement in a physical system is not under discussion here. At this point, the components are meeting demand without excess flow or heat transfer in the loop, so increasing system efficiency can only be achieved by increasing the efficiency of individual components.

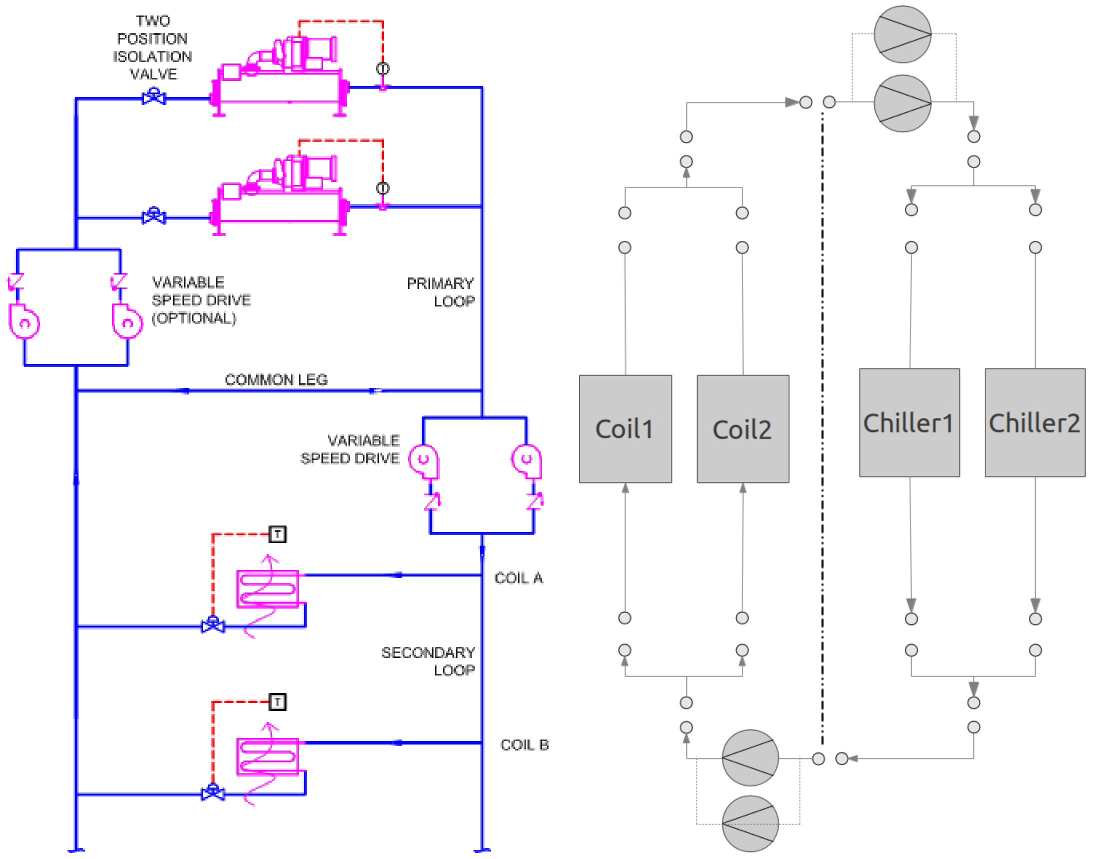
2.4.1.10 Conclusions

Abstracting from a physical system to a simulation model required initially analyzing the system topology, followed by a selection of components, followed by implementation of controls. By selecting controls and components exactly to match the physical system, a functional simulation model resulted, but did not operate as one would expect. The system was inefficient, with extra components running and excess flow around the loop. By taking time to implement controls based on an analysis of the intention of the system, an improved abstraction resulted that operated much more efficiently. Finally, a test study was performed that added energy efficiency to the system using variable speed pumps. This showed further energy savings, but was not a part of the original physical system.

A direct path from physical systems to simulation models may not exist, but instead requires a deeper analysis of the *intention* of the system to provide an accurate and suitable simulation model. In the next section, the procedure is repeated for a slightly more complex system. The level of detail will be reduced as substantial detail was included in the current section, which can be used as a reference.

2.4.2 Case B

The following demonstration is performed similar to case A, only with a different loop topology that could be considered more complex. The physical system schematic is shown in Figure 2.19a. As with the previous loop, there are two coils, and two chillers. The most dramatic change with this loop is the pumping setup. Instead of each chiller having a dedicated pump, the supply side shares a common pumping setup, which is shown as a bank of headered pumps, each with a check valve to avoid recirculation, and the option of variable speed drives. The demand side also has a dedicated pump bank which will be variable speed to meet the demands of the coils. The mismatch of flow between loop-sides is balanced by a common leg. This has the benefit of allowing each side to run at a distinct desired flow rate to minimize excess flow and maximize energy savings. The chillers are again controlled to an outlet fluid setpoint temperature. The chillers have isolation valves to ensure that the flow is directed to the proper chiller. The coils are also controlled to a zone air setpoint temperature. In this case, the coils do not have a local bypass leg to meet the setpoint, instead relying on a simple valve along with the variable speed loop pumping to meet demand properly.



(a) Physical system schematic, courtesy of Steve Taylor (used with permission)

(b) Simulation Loop Schematic

Figure 2.19: A constant speed pumping system with multiple chillers and multiple coils; dedicated pumps

2.4.2.1 Topology

In this physical system, the distinction between demand and supply sides is again clear, with the demand side pumps and coils translating directly into an EnergyPlus loop-side. The supply side is similar, with supply side pumps and chillers. Banks of pumps are considered as a single object inside the simulation, so they need only be represented by a single component (though they are drawn into the simulation schematic for clarity). The interesting difference in topology for this loop is the addition of a common pipe. This allows the demand and supply sides to operate at different flow rates concurrently to maximize efficiency. In the simulation, the common pipe is not a distinct object, but rather in inherent optional part of the core simulation topology. The resulting simulation topology is shown in Figure 2.19b.

2.4.2.2 Components

The component models used in this case are the same as the previous case for the coils (load profiles) and chillers (air-cooled). The pumps are different in that they are now pump banks. These objects consist of any number of similar pumps working in parallel. Pump staging can be performed uniformly or sequentially, and the pump bank can be variable speed. In this case, the total pump bank flow rate is varied, and the individual pump operation is determined after the total flow rate is determined.

2.4.2.3 Controls

The physical system is controlled in a similar fashion to the previous loop. The difference on the demand side is that the coils do not have local bypasses to control to a zone air setpoint, but instead have a simple valve and rely on the variable speed pumping to meet the varying flow requirement. In the simulation, this is the same as Case A. The coil components still request a flow in order to meet the setpoint. In the previous case, the only pumps were on the supply side, and so these pumps respected the demand and supply flow requests in order to meet the overall loop demands. In this case, the demand side has dedicated pumps, so only these pumps will be used to meet the coil demand. The supply pumps will operate to meet the supply side requests. Note that these supply side requests are implicitly responding to the heat transfer demands imposed on the loop by the coils. The supply side now consists of a common pumping bank, instead of dedicated pumps for each chiller. While the physical system schematic shows the chillers operating to an outlet setpoint temperature, the previous section described in detail the problems encountered with this control type in the simulation model. Instead of re-discussing these problems, this system will be simulated to a loop setpoint to bring the controls to a higher level and allow the chillers to instead respond to a dispatched load, rather than running specifically to meet an outlet setpoint temperature.

2.4.2.4 Demonstration and Analysis

The physical system was modeled in EnergyPlus, again using load profiles to perform the demand and flow requests that would be encountered by the coils. Pump banks are simply lumped pumping objects in EnergyPlus which post-process the required flow rate to determine how many of the headered pumps should run. Thus, the first case run includes a simple single pump object for each loop-side, with a second test determining the effect of using headered pumps. Since most chiller plants will run a constant speed supply pump (to meet chiller requirements), that is the configuration selected here. The demand side is variable speed. The common pipe is set up by specifying a keyword in the plant loop object definition. This will allow flow to pass through the common pipe and balance the flow mismatch between demand and supply sides. The load profile objects were set to load the loop (both in terms of thermal demand and flow requesting) in the same manner as in the previous case.

The outputs from the simulation include the same as in the previous case, but with the addition of primary/secondary flow and, in the case case of headered pumps, the number of pumps running at a given time. For the single pumps case, the outputs are shown in Figure 2.20.

The first plot in Figure 2.20, showing load profile demand loading, is exactly the same as the previous cases, with the scheduled and actual demands matching exactly.

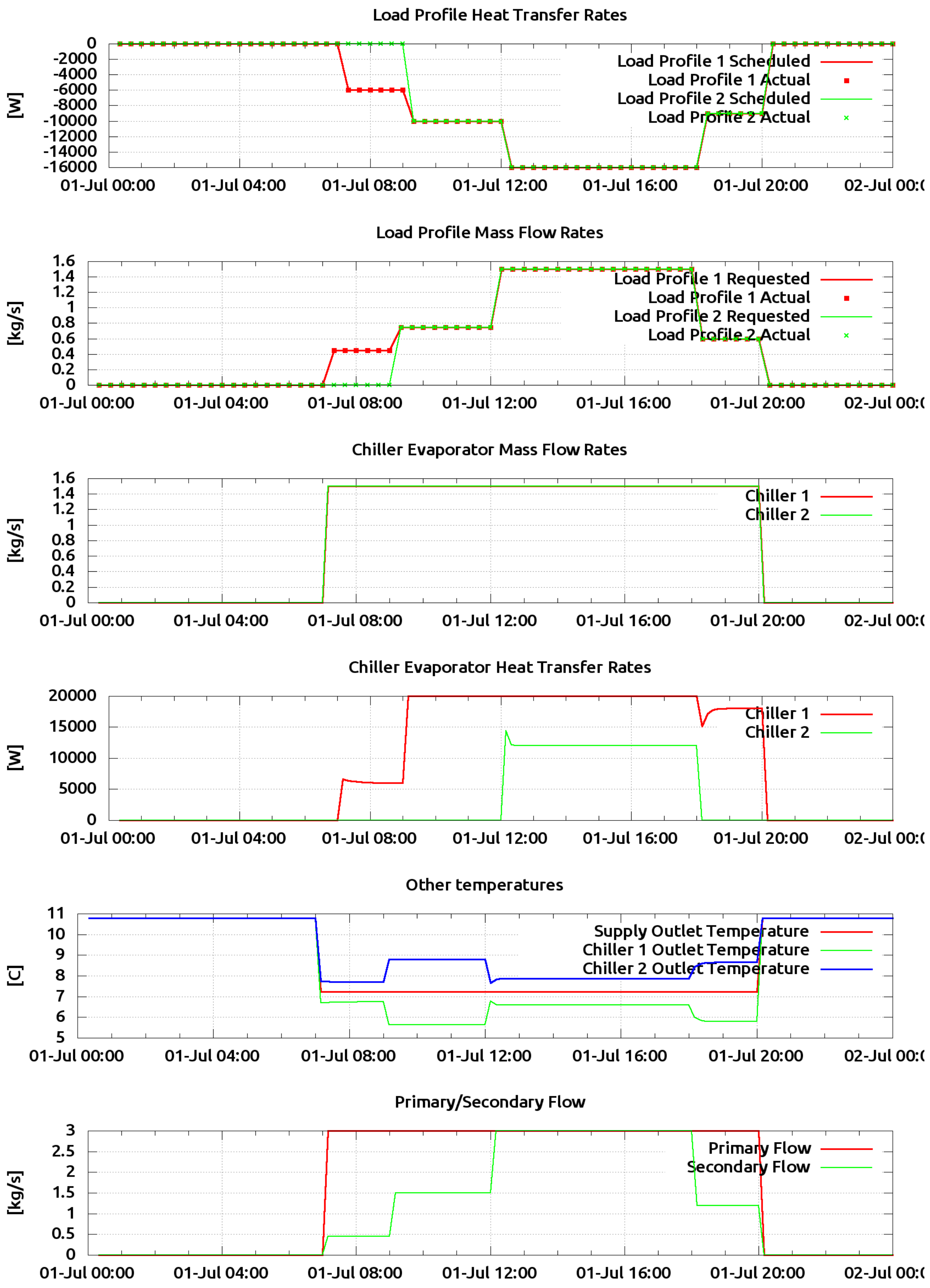


Figure 2.20: Results: Loop B: Single primary/secondary pumps

The second plot, showing load profile flow rates, shows again that the scheduled and actual conditions match exactly. This indicates that the pumping systems around the loop are capable of meeting exactly the flow requests on the loop, without starving any coils or forcing excess flow through them.

The third plot shows chiller fluid mass flow rates. The supply side pumping is constant speed, so anytime there is a request on the loop the pumping system will run at full design capacity. Thus, both chillers show a full flow rate. The fourth plot, showing the chiller evaporator heat transfer rates, shows that only the first chiller is loaded initially. In the afternoon, when the demand increases further, the second chiller gets loaded as well and starts rejecting heat, but only at a part load condition.

The fifth plot, showing temperatures around the loop, shows that the first chiller must subcool beyond the loop setpoint in order to mix back with the warm fluid passing through the second chiller, and result in a final mixed chilled water supply temperature. The general story from this plot is that no matter what the individual chillers are doing, the supply temperature setpoint is met throughout the day.

The sixth plot, showing primary and secondary flow rates shows that, again, the supply side flow rate is constant and at full design capacity whenever there is any requests on the supply side. The demand side, however, varies exactly with the load profile requests. This resolves back into energy savings in cases where the chillers must have a constant flow rate. The imbalance in flow is matched with a flow induced in the common leg.

This was simulated using single pumps for the demand and supply sides. Using pump banks has the potential to reduce pump energy further, though in many physical applications pump banks are used also to allow redundancy for maintenance and failure purposes. In the next case, the pumps are switched to pump banks, with two pumps each, as in the original system schematic. No other changes are made to the simulation. The results of this are shown in Figure 2.21.

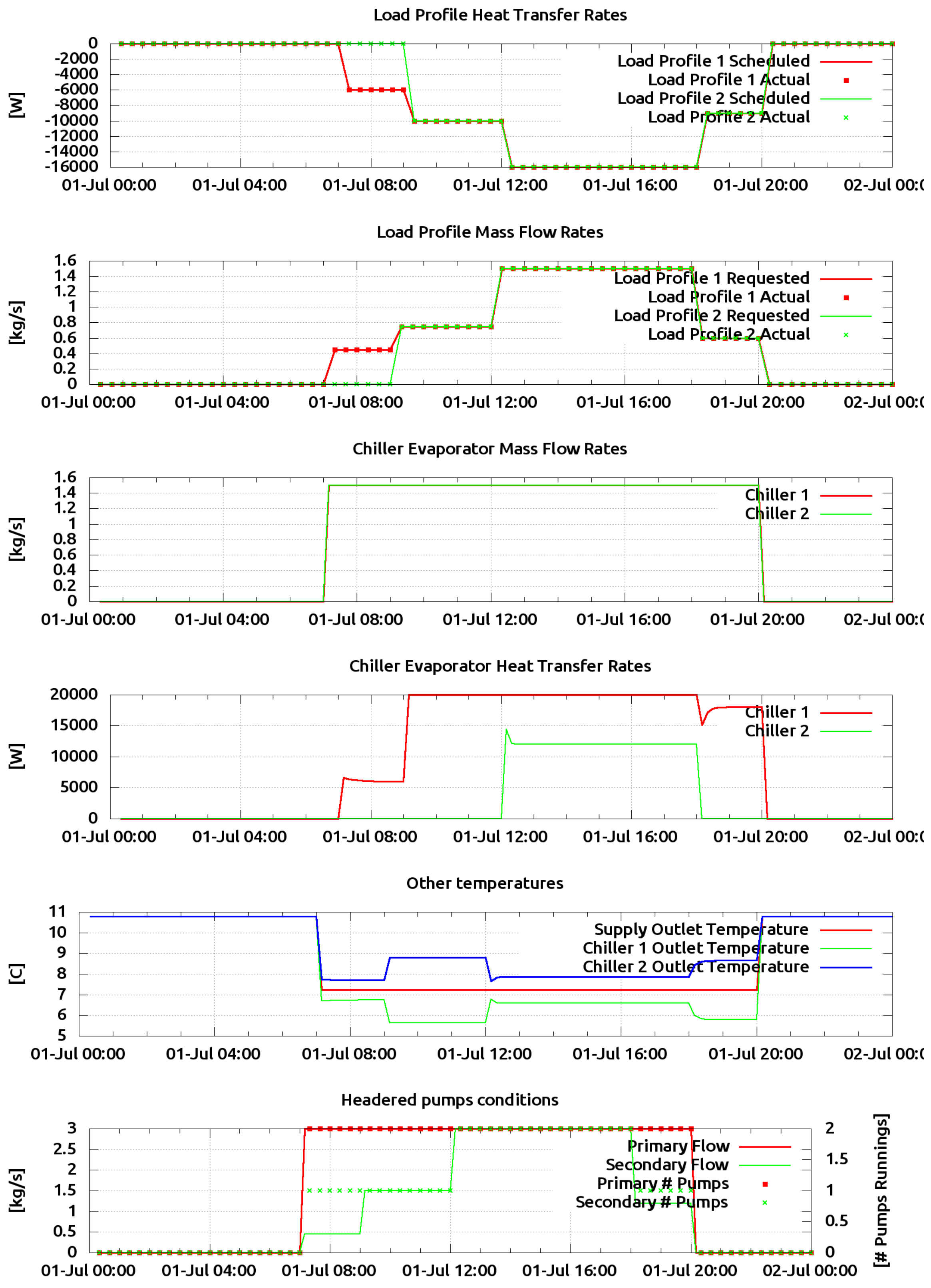


Figure 2.21: Results: Loop B: Headered primary/secondary pumps

In Figure 2.21, the first through fifth plots are identical to the single pump case. The reason for this is the way that headered pumps are simulated in this model. The pump banks are simple pump objects which are either variable or constant speed. Once the required flow is resolved, the number of pumps running in the pump bank can be calculated. Thus, the constant speed pump bank will try to run at full capacity whenever there is any demand, and variable speed pump banks will vary the total flow rate in order to track with demand.

The sixth plot shows an additional report, which is the number of pumps running in each pump bank. The supply side pumps run at full capacity, so both pumps are running. The demand side pumps show the pump bank runs with a single pump throughout the morning, even with that single pump running at a part load rate in the early loading hours. Eventually, when the load increases, all pumps run at full capacity, then trail back off in the afternoon. This could be resolved back into individual pump energy use, and perhaps show a reduction in pumping energy.

2.4.2.5 Conclusions

This second demonstration case was more concise than the first case, given that most details were similar to the first case. The model abstraction process was similar to the first case outside of the common pipe simulation and the pump banks. In order to provide an even closer match to the physical system, the constant speed pumping system would need to be improved on the supply side to respond to individual chiller demands. This leads to using dedicated pumps for each chiller which was demonstrated in the previous case.

2.5 Conclusions

A new simulation model has been developed for central plants and other fluid loops for use in the whole building energy simulation environment EnergyPlus. The simulation provides accuracy and flexibility suitable for energy analysis while reducing the computation and input burden of requiring a full network flow solution. The solution algorithm is robust in ensuring mass and energy balances are obtained, while providing a broad set of capabilities in terms of control strategies. This is achieved because the solver treats all component models as control volumes which must interact with the solver at a high level, utilizing specific interfaces to determine operating points and other boundary conditions. After a literature review and a detailed description of the solver, model abstraction studies provided confidence in the model operation and a glimpse into the process of modeling physical systems with this simulation model. The simulation model was implemented in EnergyPlus and released for public use in 2011. It is currently being used in simulation studies of buildings around the world.

CHAPTER 3

Evaluating Fluid Transport Delay in Central Plants for Whole Building Energy Simulation

Abstract

Transport delay in fluid systems is realized by four mechanisms: the time delay for transporting the fluid spatially through the system, mixing within the fluid, thermal diffusion in the fluid, and boundary heat transfer. The interactions between these factors are investigated with a review of related literature from a variety of sources. Transport phenomena is discussed within the context of whole building energy simulation. A focus is determining whether capturing such detailed physics is justified in this type of simulation environment.

Experimental measurement of transport delay is performed using a horizontal borehole field and high time-scale resolution. This experimental data is used as a validation source during the investigation of different modeling approaches in an attempt to bound the effects of delay on a system response. This preliminary modeling work is performed in a standalone application using a heat source and a discretized piping segment. Bounding calculations are provided as the basis for continuing development and implementation into a whole building energy simulation tool. The effects of varying transport model are evaluated in terms of a chilled water plant simulation case, and shown to, at least in the preliminary results, be insignificant.

3.1 Introduction

Transport phenomena within a piping system is characterized by both mechanical (mixing, transport) and thermal (diffusion and boundary) phenomena. Fundamentally, these effects are induced by the axial *and* radial variation of properties of the fluid in the system. These effects are typically blurred by the assumptions in place in building simulation programs that include fluid loop simulation. The most commonly invoked assumption is that at any single axial point in the piping system, the fluid state is uniform, and can be represented by a mixing-cup condition. This is the resulting state if the fluid at this cross-section is sampled and mixed together in an isolated cup (Tamir and Taitel, 1972), or a "mixing-cup". With this assumption, the model does not directly contain a representation of the radial property variation of the fluid at any point in the system.

The current work investigates transport delay in the context of whole building energy simulation. The effects are seen in the system response, which may have an effect on the control strategies being employed. In some applications, these effects may be insignificant, while others may show some significance. The goal of this work is to provide results from a preliminary study of this effect by modeling the transport delay in a whole building energy simulation tool and comparing against experimental data.

3.1.1 Physical Characterization

In order to distinguish between the different physical phenomena in place in a transport situation, each mechanism is investigated individually. In general, the entire physicality of the system is governed by the continuity, momentum and thermal conservation equations. For incompressible flow with no body forces, these can be modeled with the following set of equations, although this full set includes terms that are justifiably neglected in many cases:

$$\operatorname{div}\vec{v} = 0 \tag{3.1}$$

$$\rho\frac{D\vec{v}}{Dt} = -\nabla p + \mu\nabla^2\vec{v} \tag{3.2}$$

$$\rho\frac{Dh}{Dt} = \operatorname{div}\left(k\nabla T\right) + \mu\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right) + \delta_{ij}\lambda\operatorname{div}\vec{v} \tag{3.3}$$

Applying and numerically solving a system of equations such as this within a real piping system produces a representation of the physical phenomena in the system. The viscous and thermal boundary layers can be represented, which eventually cause mixing and turbulence in the fluid. For the context of building simulation, the computational burden incurred by solving such a large set of equations over a numerical grid for the sole purpose of predicting the time delay of fluid temperature in a system is not feasible, and likely not necessary.

At a larger scale, boundary layer effects can be characterized as a set of individual physical reactions. If these reactions are predicted by a modeling approach that is based, at least loosely, on fundamental physics (semi-empirical), it can provide a useful mechanism for evaluating the comprehensive transport delay effects. If fundamentals are disregarded, the interactions between effects will be difficult to isolate and capture in a model. The effects are manifested as four distinct yet interactive physical mechanisms:

- Physical delay in spatial fluid transfer through the system
- Intra-fluid axial mixing (convective and diffuse mass transfer)
- Intra-fluid axial heat transfer (convective and diffuse heat transfer)
- Boundary heat transfer

Physical delay and mixing are mechanical phenomena. Delay represents the physical time spent by, or the *age* of, a particle of fluid as it passes between two points in a system. Detailed transport delay effects are modeled using residence time theory. Mixing is the redistribution of fluid at a given point in the system, driven by turbulent activity within the flow. If we consider the mixing-cup temperature of the fluid as a representation of the bulk fluid state at a given point in a system, radial mixing will not affect the state at any axial point. However, axial mixing can result in a delay effect.

Internal and boundary heat transfer are purely thermal phenomena. Thermal diffusion is the energy exchange between particles in the system. Similar to mass diffusion, radial diffusion will not affect mixing-cup temperature, however axial diffusion will. Boundary heat transfer is the heat addition to the fluid, a convective gain from a pipe to a fluid directly adjacent to the wall. In most piping system models, the addition is instantly applied to the bulk average fluid temperature at that axial point in the system.

3.1.1.1 Transport Delay

The physical delay of transporting a fluid through a system is in general a function of average flow conditions, since it is essentially a pure time lag. However, in a physical system the bulk flow condition is only a blurred representation of the detailed flow variation, including radial velocity or temperature variation in a circular pipe. These variations can be captured by residence time theory (Nauman and Buffham, 1983).

The mean residence time of all fluid entering a system is:

$$\bar{t}_{res} = \frac{V}{Q} \tag{3.4}$$

However, much more detail can be found than this average, bulk condition. For laminar, steady, one-dimensional flow of a Newtonian fluid in a circular tube, the velocity profile is known (by application and solution of equations (3.1) and (3.2)):

$$\vec{v}\left(r\right) = 2\bar{\vec{v}}\left(1 - \frac{r^2}{R^2}\right) \tag{3.5}$$

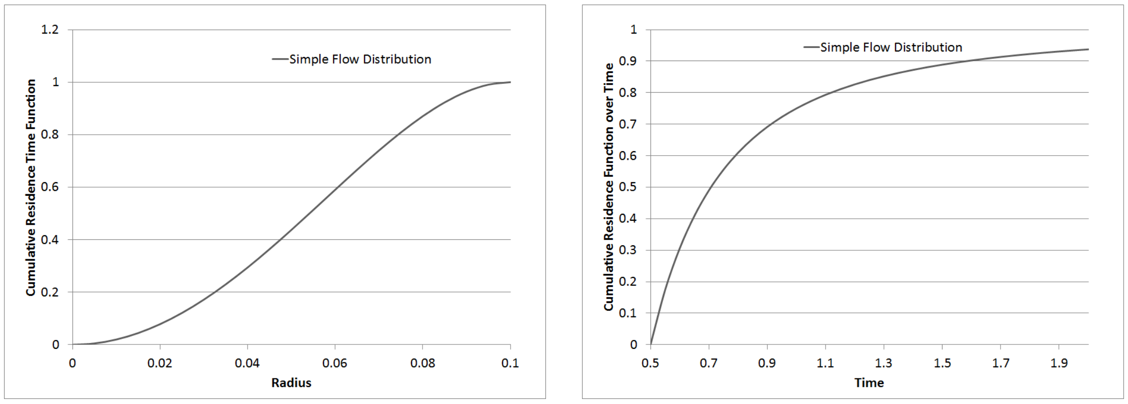
This can be used to find the cumulative distribution function (CDF) of residence time as a function of radius. The CDF expression for a monotonic velocity profile is:

$$F\left(r\right) = \frac{1}{Q}\int_0^R 2\pi r \cdot \vec{v}\left(r\right) dr \tag{3.6}$$

Integrating this velocity profile yields the residence time as a function of radius:

$$t_{res}\left(r\right) = \frac{2r^2R^2 - r^4}{R^4} \tag{3.7}$$

The residence time distribution for this velocity profile with a radius of 0.1 is shown in Figure 3.1a.



(a) Function of radius (b) Function of time

Figure 3.1: Cumulative residence time distribution under simple flow conditions

Consider that Figure 3.1a shows a normalized measure of how much of the total flow has passed through the system at a given point in time based on the position of a radial sweep over the pipe cross section. The result becomes more useful when transformed to the time domain. This is achieved by eliminating the radius term from equation (3.7) and the following relation which is reformulated from equation (3.4) for a flow that is highly one-dimensional ($L \gg r$):

$$\bar{t}_{res} = \frac{L}{\bar{v}} \tag{3.8}$$

The resulting time based cumulative distribution function is:

$$F(t) = 1 - \frac{\bar{t}^2}{4t^2} \tag{3.9}$$

Note that equation (3.9) is only valid for times when the flow has actually been transported through the entire system, or when the cumulative distribution function is positive (for this case $t > \bar{t}/2$). This value represents the first appearance time of the fluid. The resulting distribution is shown in Figure 3.1b for a system of a particular length.

With a no-slip condition at the wall using the prescribed velocity profile, as the value of radius approaches the wall, the residence time becomes unbounded. This is shown with the asymptotic behavior of equation (3.9) ($\sim t^{-2}$). Thus, as expected, the molecules directly adjacent to the surface have an infinite residence time.

The preceding analysis is used to demonstrate the manner in which overall transport delay in a fluid system is described using residence time theory. In general, this is useful for determining the time a mass will spend reaching a point downstream. This model is based on fundamental physics under given assumptions, without including any mixing of the fluid within the system. The addition of mixing or diffusion within a fluid of different concentrations (including energy concentration and therefore temperature) introduces another layer of analysis.

3.1.1.2 Mixing

As fluid is transported through a system, not only thermal gradients but also momentum gradients in the system tend to diffuse over time. This dampening then appears as a delay in transporting a perturbation through the system. Ekambara and Joshi (2003) described the effects of mixing in turbulent (and transition) pipe flows. The mixing occurs at different scales in the system: molecular, eddy, and convection. The concentration of a material within a fluid stream is governed by a transient mass balance equation:

$$\frac{\partial c}{\partial t} + \frac{1}{r}\frac{\partial}{\partial r}\left(rc\vec{v}_r\right) + \frac{\partial}{\partial z}\left(c\vec{v}_z\right) = \frac{1}{r}\frac{\partial}{\partial r}\left(rD_m\frac{\partial c}{\partial r}\right) + \frac{\partial}{\partial z}\left(D_m\frac{\partial c}{\partial z}\right) \tag{3.10}$$

Where D_m is a mass diffusion coefficient, and c is concentration.

With a turbulent flow, the concentration or any other property is best represented by the bulk average property value. Removing the turbulent variation terms using Reynolds averaging procedure, the governing equation is offered as:

$$\frac{\partial \bar{c}}{\partial t} + \frac{\partial\left(\bar{c}\bar{v}_z\right)}{\partial z} = \frac{1}{r}\frac{\partial}{\partial r}\left(rD_{eff}\frac{\partial \bar{c}}{\partial r}\right) + \frac{\partial}{\partial z}\left(D_{eff}\frac{\partial \bar{c}}{\partial z}\right) \tag{3.11}$$

In this equation, D_{eff} represents the molecular and eddy diffusion, while the convective mixing is driven by the concentration gradients in the system. Note that these equations are, as with (3.1), (3.2) and (3.3), likely not useful in detailed modeling for whole building energy simulation, however it is important to describe this phenomena to understand the driving gradients in the system.

Special flow characteristics are important in evaluating the mixing in a fluid flow. The velocity profile in the boundary layer is of special importance. Assuming the universal velocity profile $\left(u^* = \sqrt{\tau_0/\rho}\right)$ may introduce significant error, as the axial dispersion coefficient is dependent on this velocity profile.

While it is understood that a fundamental model of boundary layer induced fluid mixing phenomena must contain a detailed representation of the boundary layer, empirical methods may also be able to capture mixing effects without requiring the level of detail necessary to solve boundary layer momentum equations. This is important to consider as the ultimate goal of this work is within the context of whole building simulation.

3.1.1.3 Fluid Heat Transfer

Thermal gradients in the flow tend to induce thermal diffusion in the same way that velocity gradients induce mixing. In heat exchanger applications, where turbulent flow is expected, the velocity and temperature profiles will be relatively flat compared to laminar conditions. The thin turbulent boundary layer which drives mixing in the system is not captured when using the mixing-cup approximation. The thermal boundary layer is governed by the conservation of energy equation, which is coupled to the conservation of momentum equation. The same discussion applies to thermal diffusion as mechanical diffusion. In order to capture the physicality, the boundary layer must be represented in some fashion, either fundamentally or empirically.

3.1.1.4 Boundary Heat Transfer

The addition of energy to the fluid flow occurs at the boundary layer, and the energy is then transferred to the bulk of the fluid as it flows downstream. The overall heat transfer to the fluid is generally modeled with Newton's law of cooling:

$$\dot{q}^{''} = h\left(T_{surf} - T_{\infty}\right) \tag{3.12}$$

Where the convection coefficient h is a function of the flow regime, fluid properties, and system configuration:

$$h = \frac{Nu\, h}{k} \tag{3.13}$$

$$Nu = f\left(Re, Pr\right) \tag{3.14}$$

As the fluid flows downstream through other components, the actual temperature in contact with (for example) the heat exchanger surface may be significantly different than the average temperature of the fluid over the cross section. The specific fluid flow and system conditions dictate the importance of this deviation. With a mixing-cup temperature representation, the heat addition must be added directly to the entire radial cross section, without a diffusion effect. This is not a claim that the mixing-cup approximation is invalid, only to say that this approximation does blur the actual physical processes in the system.

3.1.2 Applicability to Building Simulation

A rigorous treatment of transport is expected to be of low value within a whole building energy simulation program, especially if it comes with a significant increase in computational burden or required input parameters. The mixing-cup state point assumption in place in fluid loop simulation renders a detailed physical transport model difficult or useless as ultimately the detailed transport phenomena would become clouded by an overall axial state condition. For this work, the transport model must then balance capturing detailed phenomena while still being useful under an axial state, or mixing-cup representation. The transport effect is expected to become more significant for systems with longer pipe runs, though the effect is dependent on not only the piping length, but also the pipe diameter and flow rate. Boundary heat transfer and physical transport effects can increase with increased length, and intra-fluid mixing and heat transfer can increase with the increased diameter.

3.2 Modeling Review

The effects of transport phenomena are not isolated to thermal fluid systems. While research into transport delay of piping network provides the most directly applicable information for the current work, transport delay is also encountered and researched in industrial applications. Section 3.2.1 includes a review of transport delay in a generalized fashion, with subsequent sections narrowing toward piping system applications.

3.2.1 Generalized Transport Phenomena

Transport delay is of concern in manufacturing operations, with many operations involving heat transfer. For temperature sensitive processes, operation controls require an understanding of the effects of transport time to ensure that the desired temperature condition is achieved at given points in the system. Huang and Kung (2005) utilized transport delay in a detailed controls analysis of an electro-hydraulic servo system. Miles (1975) described a simple method for implementing transport delay in process monitoring. Mosca and Zappa (1984) developed a plant controller for use in situations where the transport delay in the system is not easily defined by step change responses. Hearns and Grimble (2010) created a model of a material which lost heat during production as it was transported over a distance. The production of the material relied on a specific temperature at the end of this transport. Over this distance, the model lost heat at a variable rate, so the speed of the material transport was actuated according to the difference between modeled and measured material temperature. The model was highly controls-based, and was developed solely for the Laplace domain. The model operation was compared to theoretical derivations and experimental data and provided a stable and optimized controller operation.

3.2.2 Fluid Systems

While generalized transport systems provide insight into alternative means of modeling the phenomena, research into the transport effect of fluids in piping systems is more directly applicable to this research work. The following discussion focuses on this fluid system related research.

Tobias (1973) used a simple analysis of a segment of pipe (or tube in a heat exchanger) and developed the governing equations for bulk fluid temperature at any point downstream of the inlet. The Laplace transform representation of this, which included the time delay, was difficult to invert to the time domain under general circumstances. However, an approximation of the governing equation provided a time domain result. The approximation was compared to the original exponential form, with error that may be significant, depending on the flow system characteristics.

Clark et al. (1985) provided a simple transport delay model for use in a dynamic modular simulation program. The model was based upon a discretization of a pipe into a number of segments, and used the bulk fluid velocity to determine an amount to shift the fluid in the pipe. A fifth order polynomial was used to represent the fluid temperature in the pipe, and linear interpolation was employed to find the temperature between any of the discrete points being solved. The model was able to predict the transport effects in a pipe when compared to experimental data. This approach clearly blurs the physicality of the transport delay, however it still captures a transport effect. The fixed fifth order transport effect is suitable for many applications, but as applications get more diverse, such as in very long piping systems, the assumed shape is likely too restrictive to be a useful approach.

Hanby et al. (2002) utilized residence time distribution, assumed a turbulent velocity profile with diffusion, and calculated a cumulative distribution function for the flow. This benchmark was used in comparing a new approach where the pipe was discretized into a number of well-mixed segments, using finite difference techniques.

The new approach was able to accurately predict outlet conditions of a prototype plant when testing using an inter-model comparison and experimental data. Hanby claimed that for a turbulent velocity profile with diffusion, the number of cells to use in discretization was optimally 46. This generalized, validated model is a suitable candidate for whole building energy simulation applications.

Brazkeikis (2010) used a fundamental approach to create a simulation model of a transient heat carrier in a conduit. The situation simulated was a valve suddenly opening with hot fluid being introduced into the system. The author contrasted the approach with the approach by Hanby et al. (2002), stating that the delayed exponent effect is not often seen in the boiler system under investigation. This work was only simulation-based, and used the electric circuit simulation software OrCAD.

Modeling transport in fluid systems need not be related to piping networks. Fujieda and Ohyama (1985) modeled the fluid delay in supplying a carbureted engine with fuel. Different conditions were considered related to how the fuel was delivered, and where the delay actually occurred. Experimental measurements were used to validate the model with a high degree of accuracy, however the model is too detailed to be suitable in the general shell of whole building energy simulations.

3.2.3 Mixing in Fluid Systems

A significant underlying phenomena involved in transport delay in fluid systems is the turbulent mixing present in the flow.

Webb and Van Bloemen Waanders (2007) and Liu et al. (2011) utilized a commercial computation fluid dynamics program to study mixing in a number of piping configurations. Speetjens et al. (2006) performed a detailed computational fluid dynamics development to study the effects of mixing in non-Newtonian fluids.

Levenspiel (1958) provided longitudinal mixing results for turbulent and streamline (laminar) flow. The assumptions in each regime differed, as molecular diffusion becomes secondary in turbulent flow, with the turbulent mixing dominating the mixing phenomena.

Ekambara and Joshi (2003) studied axial diffusion coefficients in turbulent flow for the purpose of studying concentration levels downstream in a system. A number of historical data sets and regressions were used and compared, and the model was highly accurate in a number of conditions.

Arakawa et al. (2008) utilized a one dimensional advection formulation of the momentum equation to study mixing of gases in pipes. Of most importance was development of the axial diffusion coefficient. Numerous formulations of axial diffusion coefficient were compared and discussed.

3.2.4 Special Discussion of Hanby (2002) Model

The Hanby et al. (2002) model is a well-mixed discretization model, utilizing a heat transfer boundary condition. The conduit is discretized into a sequence of well-mixed nodes, and the first-order energy balance is governed by the following equation:

$$\overbrace{m_f C_{p,f} \frac{\partial T_f}{\partial t}}^{\text{Transient Heat Storage}} = \overbrace{\dot{m} C_{p,f} \left(T_{f,i-1} - T_{f,i}\right)}^{FluidHeatGain} - \overbrace{h A_i \left(T_{f,i} - T_{w,i}\right)}^{ConvectiveHeatLoss} \tag{3.15}$$

This equation is a standard well-mixed governing ordinary differential equation. The equation can be discretized and solved using any number of methods.

In the source (Hanby et al., 2002), the optimal number of nodes is set to be 46 to best approximate an analytical transport response of a system with a turbulent velocity profile with axial diffusion. The well-mixed approximation blurs the transient phenomena of the system, especially axially along many discretized cells, each of which are well-mixed. Within this approximation, there is a possibility of leading toward a steady state assumption. In this case, the disconnect between solution domains (steady governing equation, transient transport response) may become quasi-steady state, and introduce a coupling that is heavily dependent on the time step of the system. The assertion that the optimal number of nodes is 46 may be due to a specific configuration of the system. However, the model was utilized in the source in an experimental validation procedure, and shown to be quite accurate. Thus, the approach is not discounted, but the selection of the number of cells likely varies between systems.

As the Hanby model is not significantly different from a standard well-mixed model, the remainder of this work will focus on two bounding cases: plug flow and well-mixed.

3.2.5 Summary

The ultimate goal of this work is to create a transport delay model for possible implementation in a whole building energy simulation program. To ensure maximum applicability, the model will include a simplified representation of the fluid at any axial point in the system such as the mixing-cup approximation. However, as the background and literature review revealed, the radial variation of properties provides the mechanisms for much of the transport phenomena, so this is considered carefully.

3.3 Experimental Measurements

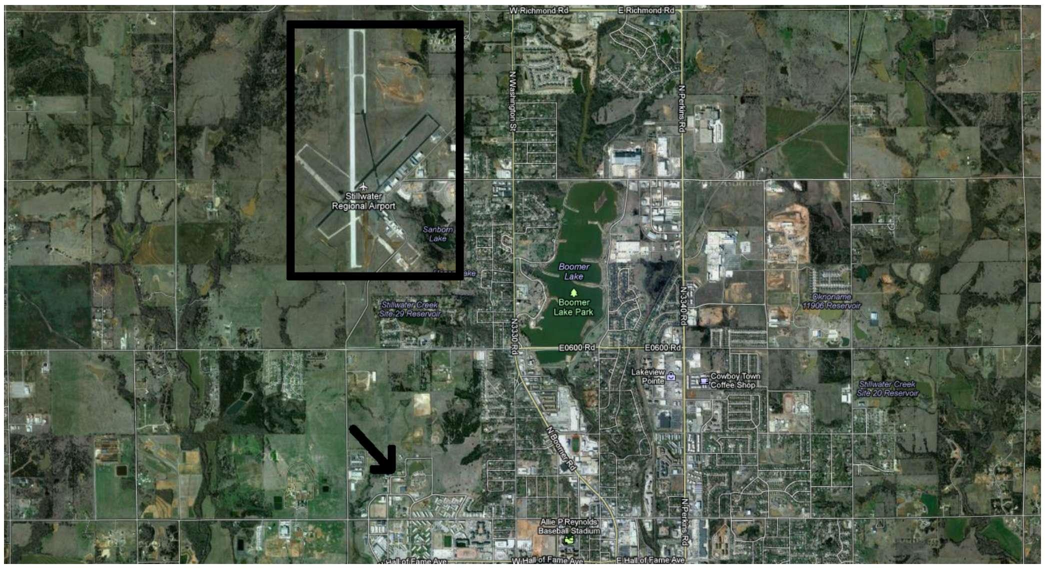
A data set was created which captures the transport phenomena at a highly refined time scale, in order to facilitate interpreting the bulk transport effects into the different physical mechanisms, that were described in section 3.1.1. The experimental site includes a series of horizontally drilled boreholes. These were drilled in 2010, and thermal response tests were performed. In the summer of 2012, data sets were created from a series of new thermal response tests on multiple boreholes. The experiment was performed under the supervision and support of Dr. Richard Beier and Bill Holloway. Although the data from 2010 did not include a high level of time resolution, comparisons are made between the new experimental data against the data from two years prior.

3.3.1 Experimental Setup

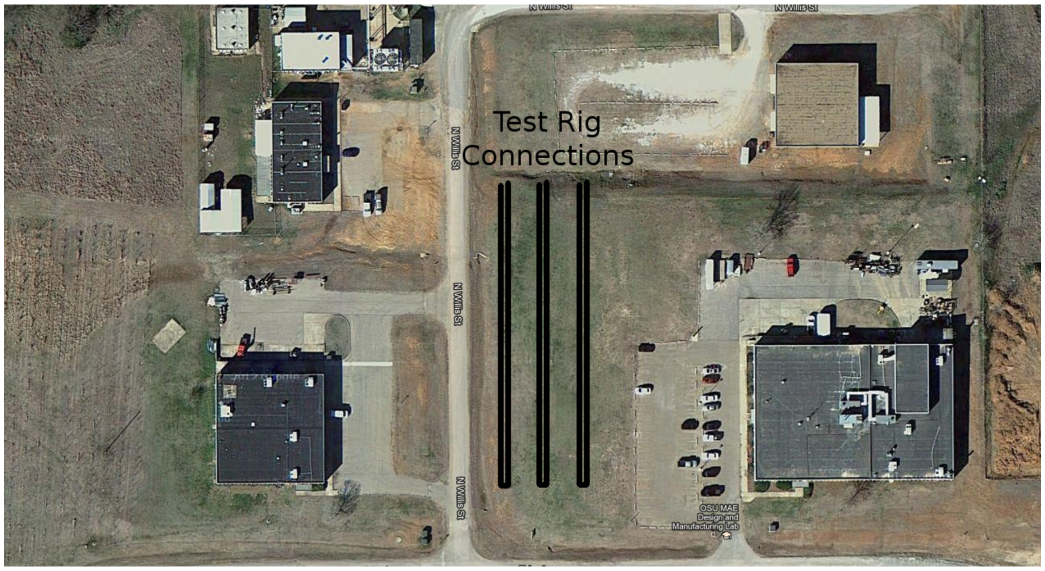
The experimental site is located in Stillwater, OK, on the campus of Oklahoma State University. As mentioned previously, the test site was setup in 2010 with a total of ten horizontal boreholes drilled in parallel. The field containing the boreholes is shown in Figure 3.2. Figure 3.2a shows the location of the field with an arrow in relation to the Stillwater Airport, which is boxed on the image. Figure 3.2b shows a closer view of the site, with boreholes drawn roughly on the image. Three boreholes are displayed on the diagram to show the orientation, however there are actually ten installed in the ground. As labeled in the figure, the connections for the borehole are exposed on the north side of the field. This is where the test rig is hooked up to each borehole during testing .

The ten boreholes at this experimental site are drilled horizontally through clay soil. The boreholes are nominally 200 feet long, with 3/4 inch HDPE SDR-11 pipe, so the total pipe run for a single borehole is 400 feet. Seven boreholes are 4.5 inches in diameter while the other three are 5.5 inches. Six boreholes were injected with Bentonite grout and cuttings during installation, while the other four were not back-filled (essentially containing just drilling mud). The boreholes were originally tested for thermal conductivity in 2010 directly after the installation. After sitting for two years, they are being retested to support this transport delay investigation and to determine if the thermal properties have changed. The boreholes with grout were expected to perform similar to the initial test, while the un-grouted boreholes were more likely to have significantly different thermal properties as seepage and movement will have occurred.

As the horizontal drilling was being performed, the depth below the surface was



(a) Broad view of test site



(b) Closer view of site with simplified borehole layout shown

Figure 3.2: Map view of test site location and configuration, provided by Google Maps positioned to longitude, latitude $(-97.082°W, 36.133°N)$. Imagery Copyright 2013 GeoEye, Texas Orthoimagery Program, Map data Copyright 2013 Google.

measured at multiple points along the borehole length. This is shown in Figure 3.3 for two example boreholes. The burial depth varies within approximately 1.5 feet along the length of the borehole for any individual borehole. Average depth between individual boreholes varies by several feet. During installation, a layer of material was found at a certain depth which was difficult to penetrate with the drilling machine. The shallower boreholes were a result of avoiding this layer.

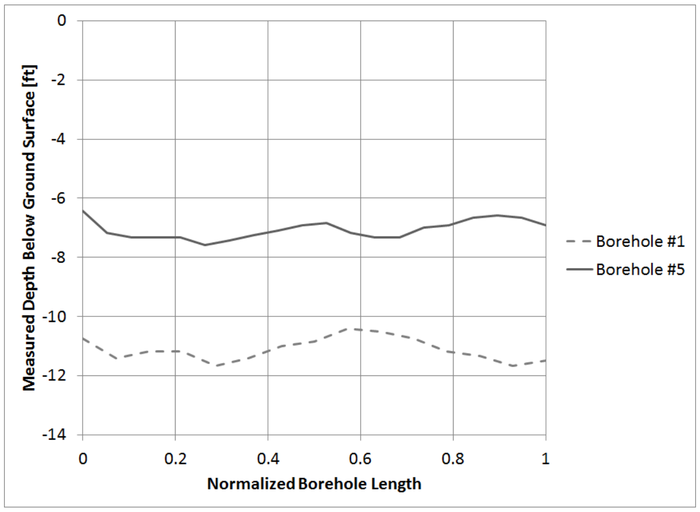


Figure 3.3: Variation of drilling depth along the borehole axis for two boreholes

For borehole modeling purposes, the average depth may be utilized, though this could add to the uncertainty of the model. If additional accuracy is desired, a method of accounting for the depth variation may be implemented. For this short time scale transport delay study, the depth of the borehole is insignificant.

3.3.1.1 Test Rig and Instrumentation

The test rig is an insulated box containing a pump, heater, and data acquisition equipment. Figure 3.4 shows one of the test rigs from two views (because there are hidden objects in each individual view) with a number of labels describing the individual components:

A: Circulation pump

B: 240 V immersed heater, originally 3500 W, reduced to 1500 W for new testing

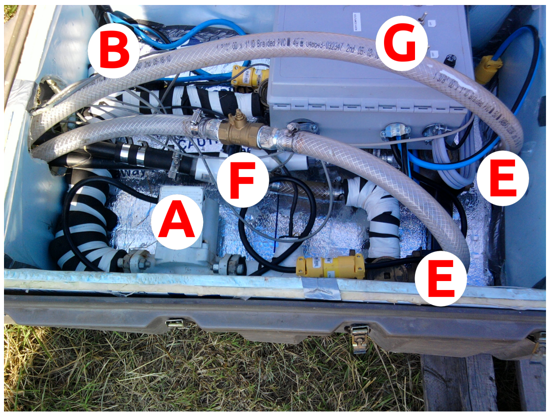
C: Temperature measurement #1

D: Temperature measurement #2

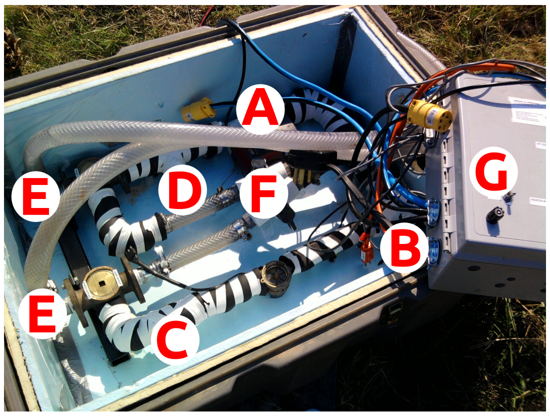
E: Connection for purge operation

F: Connection to ground heat exchanger

G: Data acquisition box



(a) View from one side of the box



(b) View from the opposite side

Figure 3.4: Layout and piping system for testing rig

Currently the measurements being taken include temperature at the heat exchanger inlet and outlet, heater power, and flow rate. A simplified view of the experimental loop is shown in Figure 3.5, with the heater and borehole elements added to visualize the location of measurements.

Undisturbed ground temperature is estimated by initiating the fluid flow without any heat addition until it reaches a steady condition. The instrumentation is capable of sampling at one-second intervals, though previous testing data was taken at a one minute interval. The program was modified to take more one-second data during the new tests to better capture the transport phenomena.

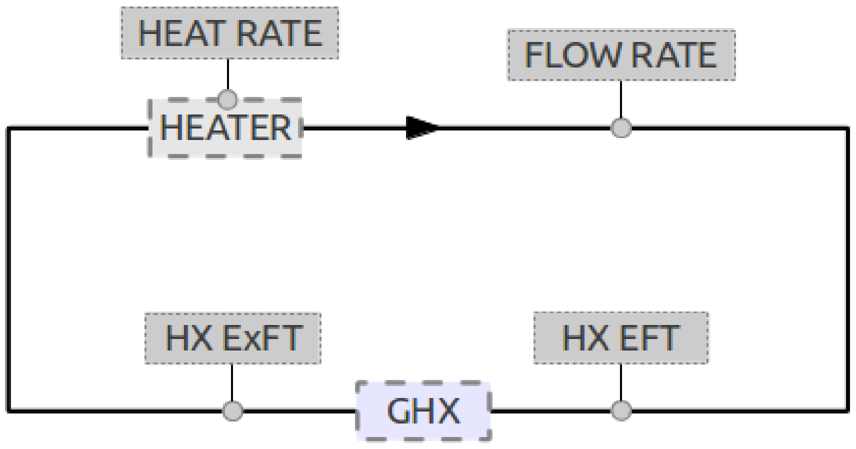


Figure 3.5: Simplified view of the experimental test rig loop data acquisition

3.3.1.2 Testing Operation

The test procedure was a standard thermal response test. In this test, the system flows continuously until a steady condition is achieved (which approaches the ground temperature at the borehole wall). The heater then turns on to a nominally constant heat rate, and the fluid and ground temperatures begin to rise. Due to transport effects, there is an initial increase in temperature of the fluid at the heat exchanger inlet, then the heat exchanger outlet. A large step increase in temperature is then encountered at the heater inlet, causing an even higher temperature gain. These steps in temperature are monitored, and once these have reached steady conditions (due to mixing, diffusion, boundary heat transfer), the transport test is complete, though the long-term thermal response test continues.

3.3.2 Uncertainty

The data set consists of a number of experimental measurements as well as assumed or approximated parameters. These are used in the simulation model to predict a system response. Experimentally measured parameters include:

- Temperatures: Inlet and Outlet (thermistor)
- Heater: Input Voltage, Input Amperage
- Flow: Volume Flow Rate

The uncertainty in these measurements can be estimated from equipment specifications or calibration data. Fluid properties (k, ρ, C_p, ν, Pr) are looked up based on average fluid conditions. The effect of using an average fluid temperature for property evaluation can cause uncertainty which can be quantified by calculating the sensitivity over a range of temperatures. The temperature variation in this experiment is not expected to require variable thermal properties.

Three physical parameters must be approximated as they cannot be known exactly:

- Pipe inner and outer diameters (may have been affected during installation or over the last two years
- Length of the system, which must include an uncertain amount of additional length to account for the test rig loop and connections
- Burial depth (measured during installation using a beacon at discrete points along the length)

The effects of running tests in neighboring boreholes will be minimized by alternating the tests and allowing a relatively long time to pass between testing neighboring boreholes. If the energy from a borehole being tested propagates to the adjacent borehole, the neighboring borehole will not be running, and the temperature of this neighbor will likely be near the undisturbed ground temperature.

Assuming an undisturbed ground temperature as a boundary for the borehole is an approximation. The spatial ground temperature distribution is not measured, so the representative ground temperature is estimated from an initial circulation of fluid in the loop. During a heated test, heat transfer will be encountered between the fluid and the ground, thus affecting the near-borehole ground temperature. An assumption can be employed that at a certain distance the undisturbed ground temperature remains. The uncertainty introduced in using this approximation can be quantified by performing bounding calculations using different undisturbed radii and also heat transfer calculations assuming ground transport properties.

The flow during the tests should be turbulent in nature, providing nearly uniform radial velocity and temperature profiles such that the effects of thermistor placement in the flow will be minimal. The boundary layer thickness and profile can be predicted assuming flow conditions (fully developed, for example), however since the thermistors are placed inside the test rig container, the flow is heavily influenced by the fittings near the thermistor, mixing further such that at this point in the system a mixing-cup temperature is a suitable approximation. Elsewhere in the system, during the long runs of heat exchanger tubing, the flow will be more developed, and the final results of a temperature measurement will still capture the overall transport delay phenomena.

Each of the individual measurements (fluid temperatures, mass flow rate), and also the fluid property values, have some uncertainty. These individual uncertainty values can be combined into a final uncertainty in derived performance values. The heat transfer rate in the ground heat exchanger system can be calculated in three forms:

Fluid Heat Transfer $\dot{q} = \dot{m} C_p \left(T_{f,out} - T_{f,in}\right)$

Heat Exchanger Surface Heat Transfer $Q = hA \left(T_{pipe,avg} - T_{f,avg}\right)$

Heater Heat Transfer $Q = IV$

Each of these forms rely on some assumptions. The fluid heat transfer relies on a mixing-cup assumption of a fluid temperature at the measurement point in the system. The area-based heat transfer rate relies on estimating an average pipe wall and fluid temperature, as well as the convection coefficient and estimated surface area. The heater heat transfer relies on an instantaneous transition from electrical energy input to heat gain to the fluid, as well as accurate measurements of the electrical values.

The uncertainty related to experimental measurements can be quantified using standard techniques such as those described by Holman and Gajda (1984). The general form of uncertainty in cases where the independent parameters undergo random, independent uncertainty w is shown here for a resultant value R and independent variables x_i:

$$w_R = \sqrt{\sum_i \left(\frac{\partial R}{\partial x_i} w_i \right)^2} \tag{3.16}$$

As a demonstration of the uncertainty expected in the experimental measurements, consider the calculation of heat transfer in the system using the fluid heat transfer expression above. The fluid specific heat is not considered in this uncertainty as the working fluid is plain water, and the temperature variation in the system is minimal compared to the documented and tabulated property data, so that the uncertainty is insignificant. The remaining two parameters are $\dot{m}$ and ΔT. In this case, the temperature *difference* is a single uncertainty rather than including the individual temperatures separately. With these two independent variables, the uncertainty in heat transfer rate can be calculated as:

$$w_{\dot{q}} = \sqrt{\left(\frac{\partial \dot{q}}{\partial \dot{m}} w_{\dot{m}} \right)^2 + \left(\frac{\partial \dot{q}}{\partial \Delta T} w_{\Delta T} \right)^2} \tag{3.17}$$

The uncertainty in mass flow rate is estimated based on an uncertainty in the data acquisition program used in the experiment. The tool reports decimal values of gallons per minute. As such, the uncertainty is estimated at 0.1 GPM. Other data could be included such as calibration curves for the equipment, however this value is expected to be a sufficient estimate. This value, when converted using an appropriate value of density, results in a mass flow rate uncertainty of 0.0063 kg/s.

The uncertainty in thermistor measurement is based on experience and discussion with researchers, and is estimated at 0.1 °C per thermistor. Since the independent variable in this calculation is not a single temperature measurement, but rather a temperature difference, the maximum uncertainty is achieved when both measurements are at maximum uncertainty in different directions. This results in a total temperature difference uncertainty of 0.2 °C.

The remaining unknowns in equation (3.17) are the partial derivative terms. For such a simple calculation these are trivial and shown here:

$$\frac{\partial \dot{q}}{\partial \dot{m}} = C_p \Delta T \tag{3.18}$$

$$\frac{\partial \dot{q}}{\partial \Delta T} = C_p \dot{m} \tag{3.19}$$

For a typical measurement, the heat transfer is calculated nominally at a value of 3643 W. The total uncertainty for this same measurement point is calculated as:

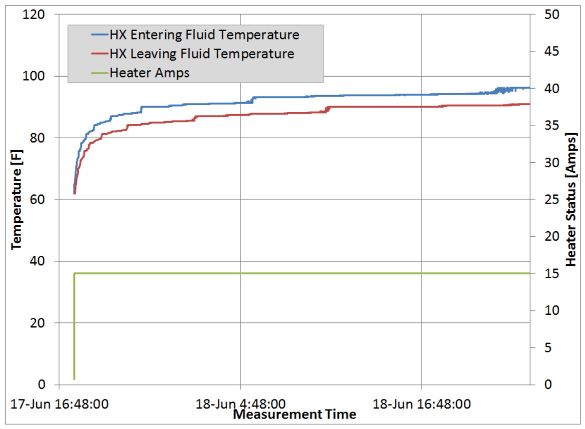
$$w_{\dot{q}} = 255.4 \text{ W} \tag{3.20}$$

This is approximately 7% uncertainty at this measurement point. As simulations are performed in the remainder of this work, this value will be considered when performing comparisons.

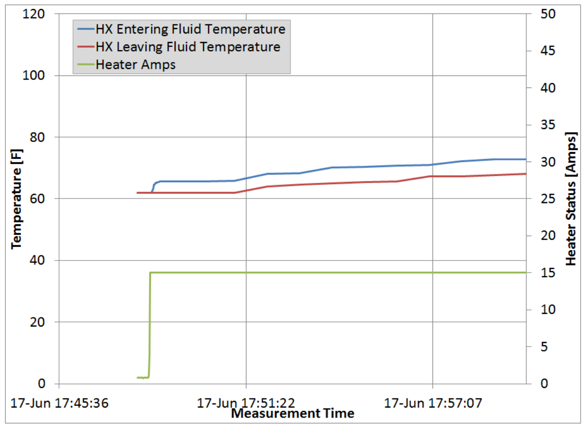
3.3.3 Measurements from 2010

The boreholes were tested in 2010 to determine the borehole resistance of the different installation configurations. While this data may not be directly relevant to the current topic, it is presented here briefly to provide a baseline and further background on the experimental site and measurements. Figure 3.6 shows experimental data for borehole #1 over different phases of a thermal response test. Note the temperature is measured at the heat exchanger inlet (entering fluid temperature) and the heat exchanger outlet (leaving fluid temperature). The heater amperage is also plotted as a signal for heater operation. Figure 3.6a shows a majority of the data set. There is an initial temperature ramp-up once the heater is turned on, followed by a progression to a nearly flat profile, with many small bumps along the way. These are the transport phenomena in the system as the warmed fluid cycles. Beyond the time scale shown in this plot, the temperature will continue increasing until a quasi-steady condition is reached where the heater heat addition approaches the heat transfer from the heat exchanger to the ground.

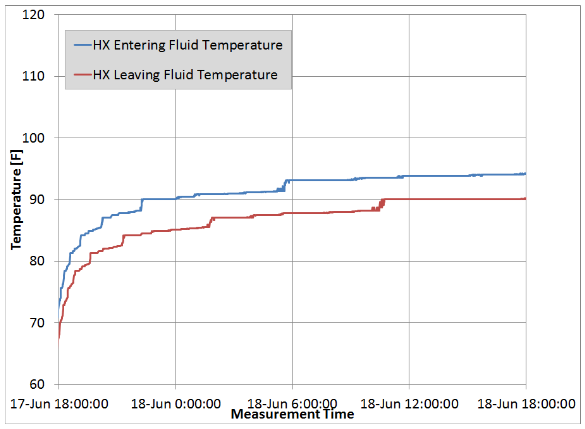
Figure 3.6b shows the detailed response during initial heater turn-on phase. As soon as the heater is turned on, the HX entering temperature ramps up to a nearly constant value over several minutes. With a nearly constant heater entering fluid temperature and heat rate, the outlet temperature of the heater will be constant also. After approximately three minutes, the heat exchanger outlet temperature ramps up as the warm fluid arrives after a full circulation through the heat exchanger. The heat gain into this already warmer fluid increases the temperature further. This stair-stepping is then seen through the rest of the data, as shown in Figure 3.6c. In these original thermal response tests, data was sampled at a one second interval until the heater was initiated, at which point the interval increased to one minute for the remainder of the test. Thus, even though this temperature measurement shows temperature rises at specific points in time, the sampling time scale blurs the actual point in time/space where the temperature increase actually existed. For a system with a roughly three minute cycling time constant, a much higher resolution is required to determine the fluid behavior. This is the justification for creating a new data set with a higher sampling rate.



(a) Large scale view of thermal response test data



(b) System temperature response during heater turn-on phase



(c) System temperature response with effects of system recirculation and transport delay

Figure 3.6: Example data from borehole #1 thermal response test from 2010

3.3.4 New Measurements

In 2012, the boreholes were retested after sitting dormant since the 2010 thermal response testing. Section 3.3.4.1 covers background information of the time that passed in between tests, and section 3.3.4.2 presents data measured in the new tests.

3.3.4.1 Climatic Activity

The original measurements in 2010 were used to determine the borehole resistance of each borehole installation configuration. The boreholes then sat unused for two years until the current experiments began. These two years were two of the hottest years on record for Stillwater, OK, breaking many temperature records including number of days above 100 degrees Fahrenheit and an all-time temperature record, with extreme drought conditions. The daily average temperatures for May through September in 2011, 2012, and also in a TMY3 weather file are shown in Figure 3.7. At first it may appear the temperatures overlay each other, however a closer analysis shows that for a majority of the season, the temperature is $3 - 5$ °C higher than the TMY3 (Wilcox and Marion, 2008) data, and peak increases of nearly 10 °C.

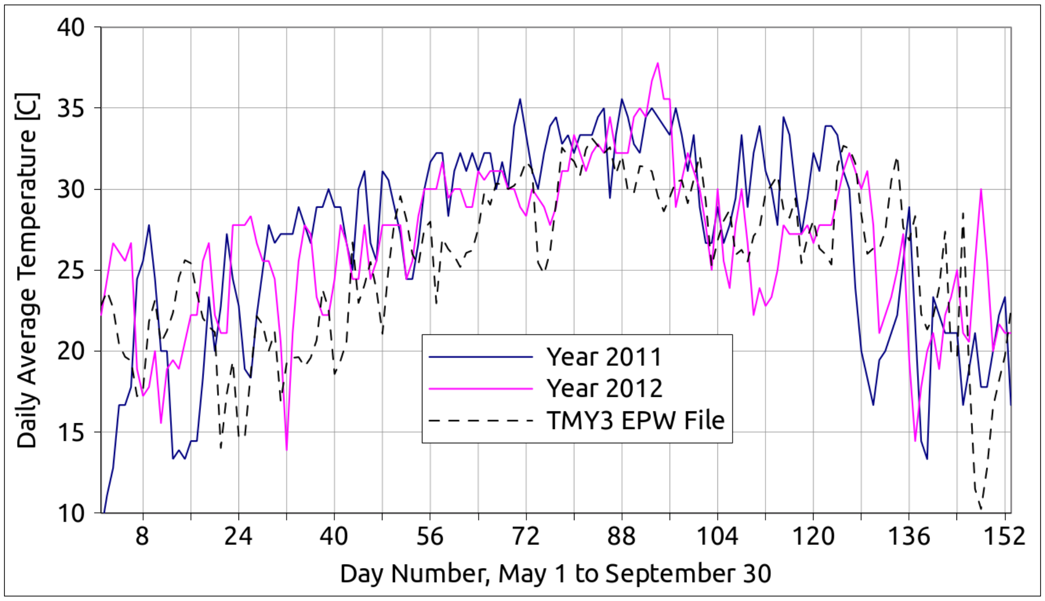


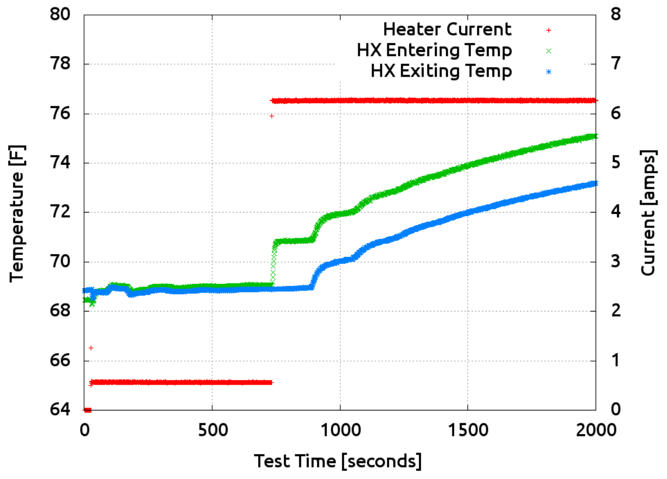
Figure 3.7: Daily averaged temperatures for Stillwater, OK in 2011, 2012, and a TMY3 weather file

During this new set of experiments, the same heater element was initially employed that was utilized in the previous tests. However, with similar flow rate conditions and heat rate, the temperatures in the boreholes quickly increased to out-of-range conditions. The conditions on the heat exchanger had changed such that it could no longer reject the same amount of heat transfer. It is possible that the dry weather and extreme heat may have affected the ground much deeper than anticipated, to the point where the soil and grout around the pipes have a much lower conductivity than originally measured in 2010. This lower conductivity effect could be the result of an air gap between the pipe and ground. In order to accommodate this behavior and still generate a usable data set, the heating element in the experimental test rig was replaced, reducing from 3500 W of heat addition to 1500 W. This change allowed the experiment to run for a full test time frame and provide useful data for the initial transport effects and also the long term thermal response test.

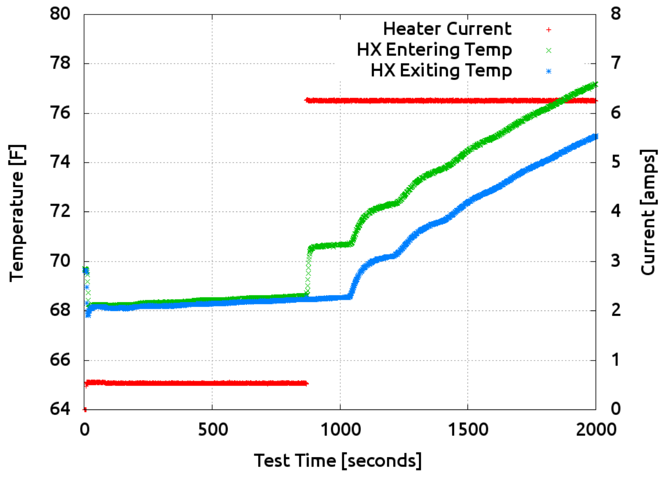
3.3.4.2 General Results

The major improvement with the new data set is the increased temporal resolution, which allows a more accurate representation of the transport phenomena, especially early in the experiment when temperature gradients are highest in the fluid. The experimental results from three different boreholes are provided in Figure 3.8. Note that these results portray the gradual transport cycling effects much better than the nearly instant stair-stepping in the 2010 data (Figure 3.6).

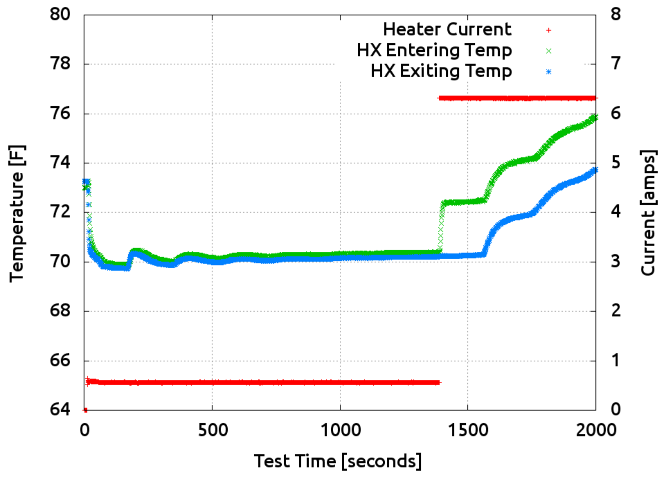
All three loops in Figure 3.8 show a similar cycling effect during the early cycles of the system, with the magnitude being governed by the thermal properties of the system. After the initial cycles, the results tend to differ. Figure 3.8a shows the response of loop 2. The temperature profile tends to flatten quickly after the initial cycles are completed. Figure 3.8b shows the response of loop 7. The temperature response of this borehole is significantly different from loop 2. This is somewhat



(a) Loop 2



(b) Loop 7



(c) Loop 8

Figure 3.8: Loops start up period demonstrating one-second data capturing

expected based on the original experimental design of 2010, as borehole 7 did not have any grout installed during the borehole installation. The borehole resistance was much higher in this borehole than in loop 2 in the new experiments. Figure 3.8c shows the diffusion of a heat impulse added to the system prior to actual test startup. In the early time values, the heater was started briefly, and a section of warm fluid cycled through the system. This heat then diffused through the fluid as well as to the pipe wall boundary.

Because the data is loop 2 is the most “well-behaved”, it is used as the main data source for performing experimental validation and other calculations moving forward in this work.

3.3.5 Comparison with Old Measurements

To conclude the discussion of experimental measurement, a brief description comparing the borehole thermal response of 2010 data to current data is possible. Consider the drastically different response in loop #7, as described in the previous section. The temperature response is much steeper than encountered in the previous testing. The borehole resistance was calculated in 2010 using a line source technique as described by Carslaw and Jaeger (1947):

$$R_{bh\#7,2010} = 0.35 \text{ hr-ft-F/Btu}\,(0.2 \text{ mK/W}) \tag{3.21}$$

However, after two years passed, the resulting temperature response yields a borehole resistance of:

$$R_{bh\#7,2012} = 0.81 \text{ hr-ft-F/Btu}\,(0.47 \text{ mK/W}) \tag{3.22}$$

The borehole resistance alone increased to over 200% of the original value. This drastic increase in borehole resistance is likely due to two contributing factors:

- The lack of grout in the original installation
- The extreme weather conditions for the two years between tests

Without grout, the ground is in direct contact with the drilling mud, which will eventually shift, leaving the ground in contact with the heat exchanger tubes. The grout is intended to stay in the borehole region, and provide a more consistent borehole resistance between the pipes and the ground.

Further investigation is not warranted in the current experiment, as the transport phenomena is isolated to the very early stages of the experiment, when the heat transfer to the ground is less than in the later stages of the test. However, the results are highly interesting in terms of research in general. It is possible that the burial depth of these boreholes is too shallow to operate under the encountered weather conditions. This indicates the need to further research suitable burial depths that are satisfactory to operate under a variety of conditions.

3.4 Simulation Testbed

The major contributions of this work to transport delay research as a whole include:

1. Creation of an experimental data set suitable for evaluating transport delay in a piping system
2. Evaluation of transport delay simulation models and propose modifications or new modeling solutions as necessary
3. Investigation of the effects of transport delay in whole building energy simulation

To begin the simulation work, a simulation testbed was developed (source code is available as appendices in this document). The testbed is a pipe simulation program which allows a variety of transport phenomena and heat transfer calculations to be utilized individually, and provides output including temperatures of the fluid along the length of the pipe. This testbed provides a model development platform where a model can be tested in an isolated environment before attempting implementation within a larger simulation program. The simulation loop used in this testbed is represented in Figure 3.9.

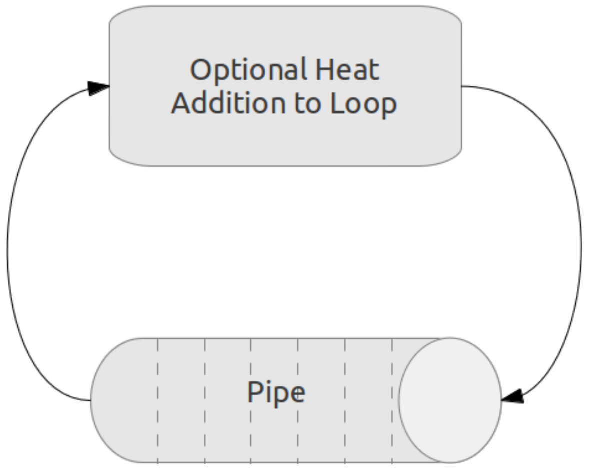


Figure 3.9: Simplified loop representation for the simulation testbed

The model uses a numerical segment by segment discretization for the pipe, and allows a heat pump model to be simulated between the pipe outlet and the pipe inlet to provide a system simulation model.

3.4.1 Program Overview

Within the testbed, the pipe is discretized into a number of segments. This number of segments is an input parameter. The model uses a uniform segment length:

$$L_{seg} = \frac{L_p}{N_{seg}} \tag{3.23}$$

For pure mechanical transport, the residence time for the entire pipe can easily be calculated from a known bulk velocity:

$$t_{res,p} = \frac{L_p}{\vec{v}} \tag{3.24}$$

These conditions are used to select a time step that conveniently matches the pipe spatial discretization with each segment's residence time:

$$\Delta t = \frac{t_{res,p}}{N_{seg}} \tag{3.25}$$

If heat transfer calculations are performed, the heat transfer is defined from the fluid to the outer pipe wall surface. In a real system, the heat transfer phenomena extends well beyond the pipe outer radius, however the effect of this is dependent on the timescale of the experiment and the transient transport properties of the pipe and soil. As the testbed is utilized for short time-scale transport delay, the pipe outer boundary provides suitable accuracy. The conductance from the fluid to the outside of the tube is modeled as the following series of equations, assuming turbulent flow with minimal property variation:

$$Re_D = \frac{2R\vec{v}}{\nu} \tag{3.26}$$

$$Nu_D = 0.023 Re_D^{0.8} Pr^n \tag{3.27}$$

$$h = \frac{Nu_D k}{2R} \tag{3.28}$$

$$R' = \frac{1}{h} + \frac{\ln \frac{R_o}{R_i}}{2\pi k_p L_{seg}} \tag{3.29}$$

$$UA = \frac{A_{surf,seg}}{R'} \tag{3.30}$$

The model steps in time at a uniform rate through a given ending time. There are three different settings that can be adjusted to simulate different phenomena, each of which are described in the next sections:

1. Transport Model
2. Circulation
3. Heat Transfer

The four main transport manifestations (section 3.1.1) are captured through the use of the transport model and heat transfer models described here.

3.4.1.1 Transport Model

The transport model defines the model used in calculating the flow response in the fluid. This defines the way in which neighboring segments interact mechanically, and simulates the fluid flow patterns in the pipe over time. The options for this setting are:

- Plug Flow
- Well Mixed Segments

For plug flow, the fluid moves from one segment to the next without any mixing interaction in the fluid in adjacent cells. In the simulation model, this is accomplished with a straightforward *shift* of temperature values in the simulation data structures. In many cases, this is an inaccurate representation, however it provides a lower bound on the possible axial mixing condition in the system.

For well mixed segments, the mass entering a segment is well-mixed with the fluid already in that segment before an overall temperature is calculated and sent to downstream segments. This steady, *well-mixed*, assumption will, in most cases significantly overestimate the actual phenomena. However, this provides an upper bound on the possible axial mixing condition in the system. This is equivalent to the approach by Hanby et al. (2002).

3.4.1.2 Circulation Type

The circulation type defines the relationship of the fluid between the pipe outlet state and the pipe inlet boundary condition. There are three options:

- Heat Pump

- Simple Recirculation
- Boundary EFT

For a heat pump model, the pipe outlet temperature at the end of a time step passes through a "heat pump", in which case a prescribed heat transfer is applied to the fluid to calculate the pipe entering fluid temperature for the next time step using the expected equation:

$$T_{p,in} = T_{p,out} + \frac{\dot{q}}{\dot{m}c_p} \tag{3.31}$$

Although this option is named "heat pump", an actual heat pump simulation is not performed; the heat is directly added to the fluid, without a COP or efficiency calculation. If building loads are utilized, this should be considered in pre-processing the loads.

For "simple recirculation", the outlet temperature at one time step is directly the pipe inlet temperature for the next time step. This is equivalent to a heat pump recirculation with zero heat transfer.

The boundary entering fluid temperature (EFT) approach is used for tightly controlling entering conditions where a scheduled or fixed temperature is applied to the pipe inlet at each time step.

3.4.1.3 Heat Transfer Calculation

The "heat transfer" option defines how heat transfer from the fluid to the boundary is modeled within the pipe. There are two options:

- Adiabatic
- Pipe Outer Boundary

In adiabatic conditions, the fluid never undergoes any heat transfer interaction with the surroundings as it passes through the pipe. If this is utilized with plug flow conditions, the fluid will not change temperature as it moves through the pipe, rather it will be cycled continuously.

In pipe outer boundary mode, the fluid will interact with a specified pipe outer wall temperature. This temperature is a single value fixed in time and specified directly at the pipe wall outer radius.

3.4.2 Detailed Equation Formulation

The interactions between transport model, heat transfer, and circulation type settings results in a specific set of equations for each combination. Similarities between the equations have led to a simple structure implemented inside the testbed code, however the equations are described here for each combination of settings currently implemented:

Plug Flow - Adiabatic For plug flow, the fluid from an upstream segment moves into the downstream segment as a full shift of all segments, without any interaction between neighboring segments. For adiabatic piping, there is essentially nothing to model aside from tracking the fluid through the pipe. This can still be derived by an energy balance over the cell:

$$mC_p \frac{\partial T}{\partial t} = 0 \tag{3.32}$$

Which reduces to indicate that the temperature variation of a single (moving) segment worth of fluid is not changing temperature with time. Since the shift is the only operation taking place, this is satisfied with the following segment temperature update equation:

$$T_e = T_i \tag{3.33}$$

Plug Flow - With Heat Transfer With the plug flow model still in place, the flow between segments still does not mix. However, with the addition of heat transfer to the pipe wall, some temperature variation is captured. The governing equation begins with a heat balance

$$mC_p \frac{\partial T}{\partial t} = \dot{q}_{pipe} \tag{3.34}$$

The pipe heat transfer is currently defined as the heat from the pipe outer wall surface temperature to a representative fluid temperature in the domain. In order to carefully consider the phenomena being simulated, the transport and heat transfer calculations are performed as two independent steps. The spatial shift in fluid segments puts the upstream cell temperature into the current segment. Transient heat transfer occurs on this segment without including any flow effects. Thus the heat transfer is between the upstream cell temperature and the pipe outer wall temperature. This will effectively shift the temperature then perform heat transfer. The expression for heat gain is:

$$\dot{q}_{pipe} = UA\left(T_p - T_i\right) \tag{3.35}$$

The partial derivative in equation (3.34) is discretized in time and is in reference to the mass of fluid entering the cell. Thus, using the nomenclature where a "+" represents a property at the end of the time step, the partial derivative is represented as a finite difference[1]:

$$\frac{\partial T}{\partial t} \models \frac{T_i^+ - T_i}{\Delta t} \tag{3.36}$$

Evaluating the heat transfer entering the fluid (eq. (3.35)) at the *beginning* of the time step enforces an explicit approach to the fluid update calculation. While this is not an inherently stable approach, the time step used in this model is expected to be small, and could be checked to ensure stability during simulations. The final form of the update equation is shown here:

$$T_e^+ = T_i + \frac{UA}{mC_p}\left(T_p - T_i\right) \tag{3.37}$$

[1]The $\models$ operator signifies that one operand *models* the other operand.

Well Mixed - Adiabatic For a well-mixed system, an instantaneous complete mixing occurs in the system at each cell. This is an exaggeration of the mixing effects in most piping systems, but provides the opposite boundary to the plug flow approximation. The first step in this calculation is to perform a full mixing simulation. This is governed by the equation:

$$\sum \dot{q} = 0 \tag{3.38}$$

Each of the fluids will transfer some heat during the mixing processes, and the entire mass will end up at a resulting temperature. Thus, the heat transfer for each fluid is governed by:

$$\dot{q} = mC_p \left(T_{fluid} - T_{mixed}\right) \tag{3.39}$$

Coupling this into the governing equation for both fluids (1 and 2) being mixed results in the following update equation for a mixed temperature:

$$T_{mixed} = \frac{m_1 T_1 + m_2 T_2}{m_1 + m_2} \tag{3.40}$$

Since the pipe is adiabatic, the mixed temperature is the outlet temperature:

$$T_e = T_{mixed} \tag{3.41}$$

Well Mixed - With Heat Transfer The well-mixed flow assumption, as applied in this testbed, inherently adds a steady-state nature to the fluid in order to bring about a fully-mixed condition in each segment as the fluid flows downstream. With the addition of boundary heat transfer, the option is open for utilizing a transient or steady state formulation. Currently, the heat transfer in this combination is modeled as transient. This demonstrates the decoupled nature of the mixing and heat transfer calculations for this testbed. Eventually, the heat transfer formulation could be an option for the testbed.

In this mode, the well-mixed temperature is calculated for adiabatic conditions (T_{mixed}; eq. (3.40)), followed by the heat transfer calculation. The heat transfer update equation is in the same form as that for the plug flow case (equation (3.37)), but using the mixed temperature instead of the upstream shifted temperature:

$$T_e = T_{mixed} + \frac{UA}{mC_p}\left(T_p - T_{mixed}\right) \tag{3.42}$$

3.5 Model Exercise

The simulation testbed was used for a simplified simulation of the experimental setup described in section 3.3. This is not intended to represent a full experimental validation yet, only a demonstration of the model. The testbed parameters are specified as in Table 3.1.

Table 3.1: Summary of input parameters to testbed simulation for borehole #1 test data from 2010

Parameter	Value	Units	Source
initial_fluid_temp	16.577	°C	Approximated
entering_fluid_temp	16.577	°C	Approximated
pipe_outer_surf_temp	16.577	°C	Approximated
fluid_mass_flow_rate	0.29	kg/s	Experimental Setup
pipe_inner_diameter	0.02154	m	3/4" HDPE SDR 11
pipe_outer_diameter	0.02667	m	3/4" HDPE SDR 11
total_pipe_length	121.92	m	Experimental setup
pipe_conductivity	0.45	W/mK	Incropera and DeWitt (2002)
fluid_conductivity	0.58	W/mK	Incropera and DeWitt (2002)
fluid_density	997.8	kg/m^3	Incropera and DeWitt (2002)
fluid_specific_heat	4187.0	J/kgK	Incropera and DeWitt (2002)
fluid_kinematic_visc	0.8e-6	m^2/s	Incropera and DeWitt (2002)
fluid_prandtl	7.0	–	Incropera and DeWitt (2002)
Q_heatpump	3500.0	W	Experimental Setup

The experimental data showed that the near-borehole ground temperature was 61.84°F, as the system ran to a steady condition with no heat addition. This is used as the solution's initial condition, initial entering condition, and boundary condition (which for this simplified model is fixed at the pipe outer wall). The fluid flow rate is reported in the experimental data in volumetric-basis, so the density is estimated in order to calculate a mass flow rate to be used in the testbed. The heater power is calculated as the measured voltage times amperage. A constant amount of heat is not captured by the temperatures measurements in the system due to flow fluctuations and fluid temperature variation, but as the system approaches steady state, the heat gain approaches a constant value.

Based on this system setup, the average residence time for the entire pipe is calculated as:

$$\begin{aligned} t_{res,pipe} &= 153 \text{ s} && (3.43) \\ &= 2.55 \text{ min} && (3.44) \\ &= 2 \text{ min } 33 \text{ s} && (3.45) \end{aligned}$$

This testbed demonstration uses a heat impulse mode with the heater switched on for exactly half of the fluid average residence time. This produces a "half-heated/half-unheated" fluid temperature profile in the loop.

The experimental data showed transport phenomena as seen in Figure 3.6. The fluid behavior is not fully plug flow or fully mixed, rather somewhere in between. The testbed is run here for two cases: complete plug flow with no mixing, and fully mixed flow at each node. Both cases include heat transfer to the boundary (pipe wall). This test demonstrates the ability of the testbed for perform a simple bounding calculation on the system. The results of the simulation are plotted as fluid temperature profiles along the length of the system at discrete sampled points in time:

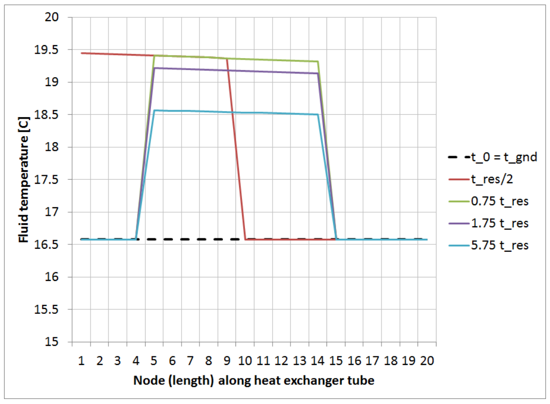
$$t_{sample} \equiv \begin{cases} t_0 & = 0 \text{ s} \\ 0.5 t_{res,pipe} & = 76.5 \text{ s} \\ 0.75 t_{res,pipe} & = 115 \text{ s} \\ 1.75 t_{res,pipe} & = 268 \text{ s} \\ 5.75 t_{res,pipe} & = 880 \text{ s} \end{cases} \tag{3.46}$$

These sample times were chosen to visually express certain conditions of the system. The initial time shows the steady condition at the ground temperature. At halfway through a pipe residence time, the front half of the system has fluid which passed through the heater, while the second half has not cycled through the heater. At this point in the simulation, the heater now switches off. At three-quarters of a residence time, the warm fluid is centered in the system, a beneficial stopping point for data visualization. Integer increases from this point show the warm fluid region centered in the system, but at later points in time.

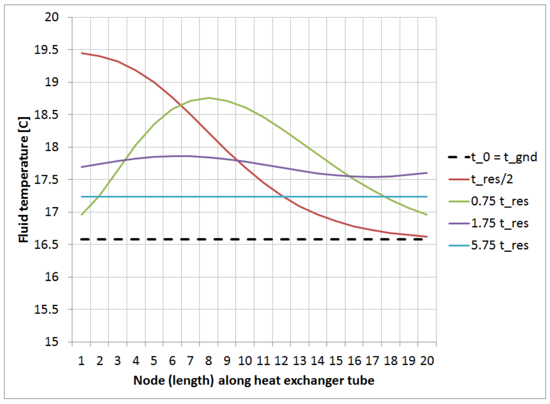
At this point, the significant level of simplification in place in this testbed is not expected to accurately match experimental data. The purpose of this exercise is to demonstrate the possibility of a simple model in capturing transport phenomena, and show qualitatively what may be possible from the experimental data.

The results of the simulation are shown in Figure 3.10. For plug flow conditions, the results are shown in Figure 3.10a, while Figure 3.10b shows the results for the well-mixed case.

The plug flow case in Figure 3.10a shows the expected discontinuity between heating and unheated fluid. The unheated fluid remains at the initial temperature, thus there is no heat transfer to the boundary (with this simple boundary heat transfer model). The warmed fluid rejects heat to the (constant) boundary over time, and



(a) Complete plug flow



(b) Fully mixed flow at each node

Figure 3.10: Simulated fluid temperature profiles at discrete times

cools. Eventually this fluid would approach the boundary temperature.

The well-mixed case in Figure 3.10b shows an unrealistic amount of axial diffusion within the fluid, such that even half-way through a residence time, the fluid in the downstream half of the pipe is affected by the heat addition to the fluid in the upstream half. This diffusion dampens the distinction between the heated and unheated fluid sections. Within five cycles, the mixing effect dampens the thermal gradient such that there is a warm mass of fluid everywhere in the pipe. This mass rejects heat to the boundary, and will approach the boundary temperature eventually.

The well-mixed and plug flow cases specify the same heat addition from the heater, since it is approximated as a constant heat rate, and both cases have the same heat addition time. However, the average temperature between the cases is slightly different as the simulation moves in time. This is because the plug-flow case has a mass of much higher temperature resulting in a higher heat transfer rate to the surroundings. The well-mixed case quickly dissipates to a milder temperature, so there is less potential for heat transfer to the surroundings. Energy is balanced in each approach, but the total heat transferred to the boundary is different, resulting in a different bulk fluid temperature in the system.

3.6 Simulation Testbed Evaluation

The model testbed was developed and shown to be capable of performing bounding calculations using a number of model configurations. In this section, the model testbed is used as a basis for investigating modeling techniques. Specifically, the model testbed's ability to match experimental data will be evaluated.

3.6.1 Data Preparation

As previously discussed, the test results from loop #2 were selected for this investigation. The results clearly depicted the transport phenomena, however the flow rate measurement for this borehole failed during this experiment. The thermal response test for this borehole was run much longer than the initial transient period, to a steady state condition for a long period. This, along with the specific measurements being made, allow for a degree of freedom for cases when a single unknown is present. The borehole steady conditions, delta temperature, and known heat gain from the heating element support a relatively accurate calculation of the mass flow rate. Figure 3.11 shows the steady state results from the loop test.

In the initial period, the borehole conditions vary dynamically. However, once steady state is reached, the electric power into the fluid from the heating element is very close to the heat rejection rate of the borehole to the surrounding ground[2]. The following nominal steady values are used:

$$\bar{q} = 1390 \text{ W} \tag{3.47}$$

$$\bar{\Delta T} = 2.097 \text{ °C} \tag{3.48}$$

Using approximate thermal properties for the water in the system ($\rho = 998$ kg/m^3,

[2]Even at steady state, there is a small variation, but this effect is averaged out by using a large sample set of the data during this steady period.

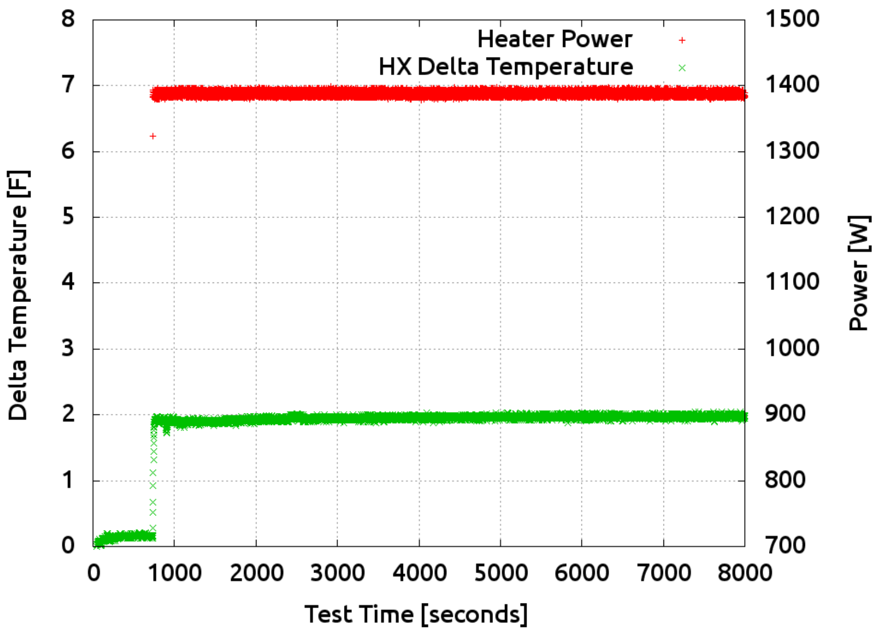


Figure 3.11: Steady state operation of loop 2 in thermal response test in 2012.

C_p = 4180 J/kgK) and the nominal flow conditions allow the mass and volume flow rates to be calculated[3]:

$$\dot{m} = \frac{\bar{q}}{C_p \bar{\Delta T}} = 0.1586 \text{ kg/s} \tag{3.49}$$

$$\dot{V} = \frac{0.1586 \text{ kg}}{\text{s}} \frac{\text{m}^3}{998 \text{ kg}} \frac{264.17\, gal}{\text{m}^3} \frac{60 \text{ s}}{\text{min}} = 2.518\, GPM \tag{3.50}$$

Although the mass flow rate in the system may vary slightly in the initial data from this steady value because of temperature variation leading to property variation leading to a different operating point, the effect is expected to be much less than the estimated overall uncertainty of 7% of total power input.

[3]The volume flow rate is not necessary for subsequent calculations but provides a reference value that may be more familiar, or relatable, to readers.

3.6.2 Comparison to Experimental Data

The simulation model testbed is exercised in both limiting flow modes: plug-flow and well-mixed. It was expected during the initial experimental design that these two modes would bracket the experimental data. The plug flow will under-estimate the mixing in the system, while the well-mixed will almost certainly over-estimate the mixing. While this will have an effect directly on the timing of the system simulation, it will also have an effect on the predicted heat transfer rates. Clearly, the heat transfer effect is highly configuration dependent, as cases with a high level of insulation will be less affected. Investigating when this is important is beyond the scope of the current task, but recommended as part of the future work of this research.

The data described and prepared in the previous section is set up for comparison with the testbed. In addition to allowing values to be hard-wired in the source code, the testbed also reads from an input file, and can also read options from environment variables set on the command line. The inputs for this particular experiment are summarized in Table 3.2. Note that many parameters in the testbed were defaults, the values shown in the table are those that override the model defaults.

The model coefficient variables (start with MODELCOEF in Table 3.2) are set to zero or one for each of the desired flow modes. The results of each of these bracketing solution are compared to the experimental response in Figure 3.12.

The simulation responses shown in Figure 3.12 are just as expected. The plug flow case shows a very unrealistic response of distinct stair step cycles throughout this entire transient period. The well-mixed case shows an overestimation of the mixing in the system, and dampens the response.

The testbed was designed to allow blending of the response of each model. In this way, when the testbed is updating a cell temperature, the resulting temperature is calculated with each model type. The model coefficient variables are then employed

Table 3.2: Summary of Input Parameters for Testbed Comparison to Experimental Data

Testbed Variable Name	Value	Units
INITIAL_FLUID_TEMP	21.27	°C
ENTERING_FLUID_TEMP	21.27	°C
PIPE_OUTER_SURFACE_TEMP	20	°C
FLUID_MASS_FLOW_RATE	0.261	kg/s
PIPE_INNER_DIAMETER	2.154E-02	m
PIPE_OUTER_DIAMETER	2.667E-02	m
TOTAL_PIPE_LENGTH	127	m
Q_HEATPUMP	1385.0	W
NUM_SEGMENTS	20	
MAX_TIME	3000	s
REPORT_FREQUENCY	50	
MODELCOEFHANBY	0	
MODELCOEFPLUGFLOW	0	
MODELCOEFWELLMIXED	1	
CIRCTYPE	HEATPUMP	
HEATTRANSFERTYPE	PIPEOUTERBOUNDARY	
HEATPUMPTESTTYPE	STEPCHANGE	

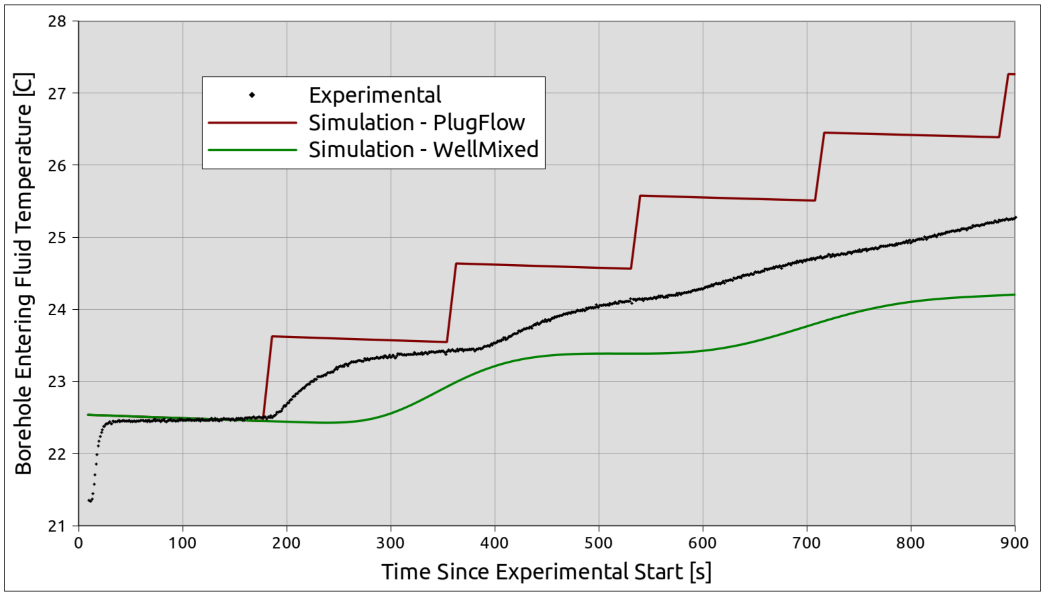


Figure 3.12: Comparison of bracketing simulation modes to experimental data

to calculate a weighted average as the final cell temperature. This allows a blended response to be achieved for the entire system. Based on the response in Figure 3.12, it would seem that a blending of 50% may result in a response that matches the experimental data well. The resulting comparison of a 50% plug flow and 50% well-mixed blend with the experimental response is shown in Figure 3.13.

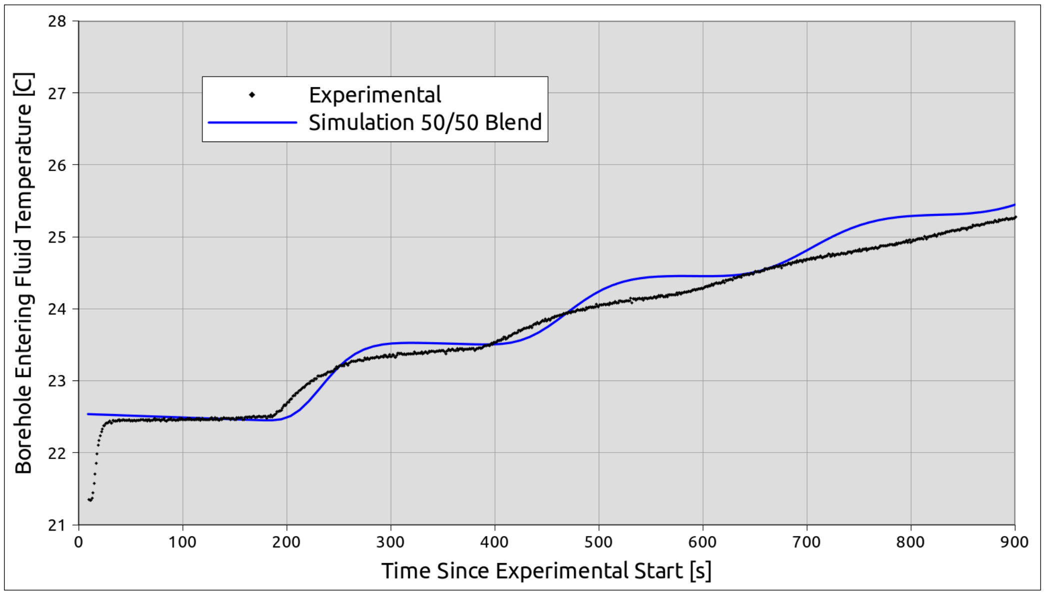


Figure 3.13: Comparison of a blended transport response model to experimental data

The comparison of the blended model with experimental data in Figure 3.13 is much better than either of the two bounding models. This is an interesting result, as this blended approach is analogous to some convection correlations, which are essentially weighted averages of results between different flow regimes. Thus, there is a precedent to continue researching this as a suitable approach, and this is highly recommended as future work. To conclude the current research work, it is sufficient to continue using a bounding case during the investigation within the whole building energy simulation environment.

3.7 Investigation in Whole Building Energy Simulation

A bounding approach is employed to conclude the investigation of transport delay effects in a whole building simulation environment. As described previously, the two bounding cases of well-mixed (or Hanby) and plug flow will provide useful information for understanding the effect. The whole building simulation tool utilized is EnergyPlus. EnergyPlus contains a detailed fluid loop simulation model (described in Chapter 2), as well as detailed interaction between zone simulation models and other aspects of the domain. EnergyPlus was enhanced with the ability to do heat transfer pipe components, as well as capture the related transport phenomena, using the Hanby model.

3.7.1 EnergyPlus Test File

The base input file used for this study is a five-zone office building served by chilled water and hot water loops. The chilled water loop is then connected to a condenser loop which is conditioned by a vertical ground loop heat exchanger. The HVAC system for the entire simulation model consists of the following:

Air Loop This loop serves the zones with chilled water and hot water coils

Chilled Water Loop This loop serves the chilled water coils with a connection to the ground heat exchanger heat pump system, with auxiliary cooling as needed

Hot Water Loop This loop serves the hot water coils with a connection to the ground heat exchanger heat pump system, with auxiliary heating as needed

Condenser Loop This loop serves the chilled and hot water loops via heat pump coils, with a vertical ground heat exchanger ultimately connecting the system to the environment.

In EnergyPlus input nomenclature, the object which simulates the heat transfer and transport delay of a pipe placed in the outdoor environment is the *Pipe:Outdoor* object. In this model, the pipe outer boundary condition is the outdoor air, coupled via a convection coefficient which is dependent on wind speed from the weather data. The other boundary condition to the pipe heat transfer model is the fluid inlet condition, which is defined by upstream components in the plant simulation model. The model then provides an outlet condition to be utilized by downstream components. The heat transfer pipe/delay object is placed on the chilled water loop in between the chillers and zone chilled water coils. This is circled in Figure 3.14, which details the contents of the entire chilled water loop.

3.7.2 Case 1: Modified Mixing Model

As determined previously, the Hanby model is equivalent to a well-mixed approach. For comparison purposes, the model was modified to alternatively use a plug flow governing equation for the discretized pipe object. To enable easy parametric studies, the EnergyPlus source code, object dictionary, and input files were modified to include an additional flag on the pipe heat transfer object which is used in the code to select the appropriate governing equation.

Both flow models, Hanby/Well-Mixed and Plug Flow, were used in two cases: a 100 m long pipe and a 500 m long pipe. In addition, a baseline case was run with no delay/heat transfer component in place.

To evaluate the effects of adding the delay object, the chilled water loop demand is reported. This will be affected by the heat transfer in the loop, as well as the delay effects. If effects are seen from the delay, they should represent an effect that will progress through the nested loop structure “downstream” and affect the ground heat exchanger conditions, and also “upstream” and affect the chilled water coils.

The results of all configurations are shown as Figure 3.15. For any configuration, the system response is much different than the base case. However, the differences between mixing models are not significant. There is no significant lag and the different

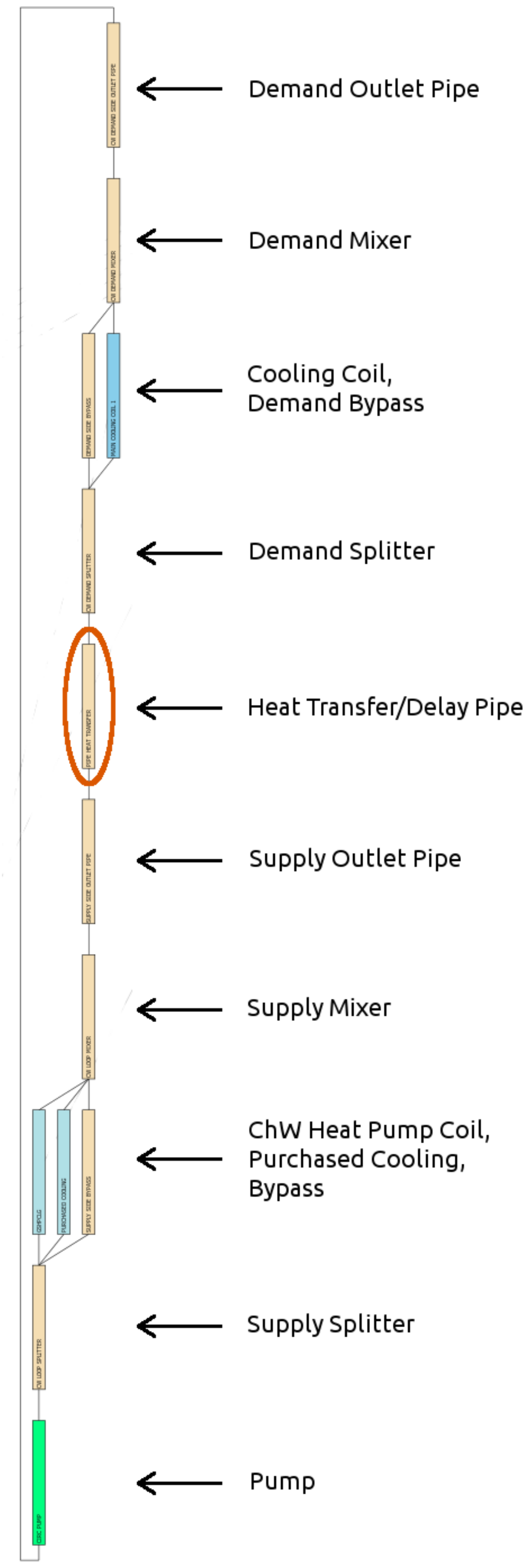


Figure 3.14: Chilled water loop topology. Delay/Heat Transfer component is circled.

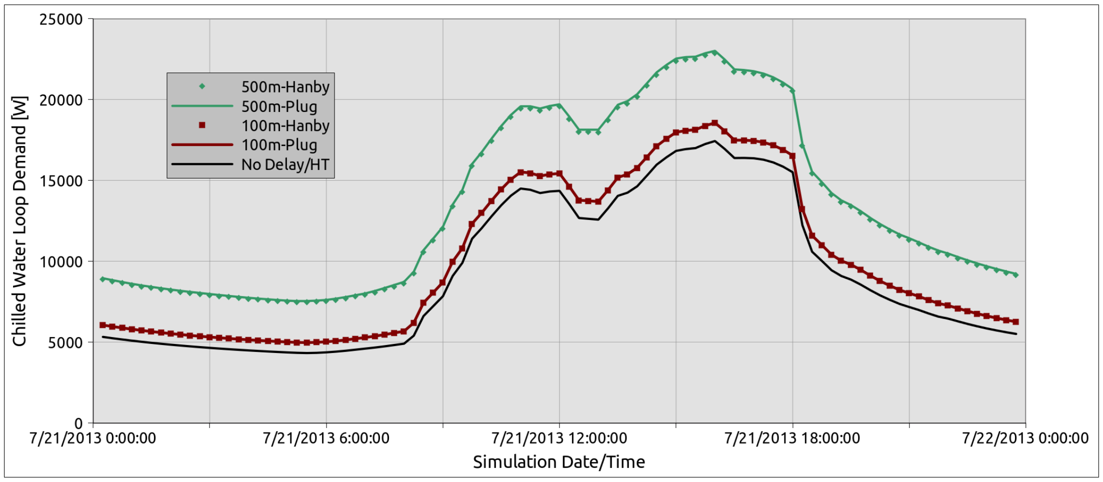


Figure 3.15: Chilled water loop demand profiles for a single simulation day under different transport modeling configurations

between well-mixed and plug-flow is present, but very small, as shown in Figure 3.16.

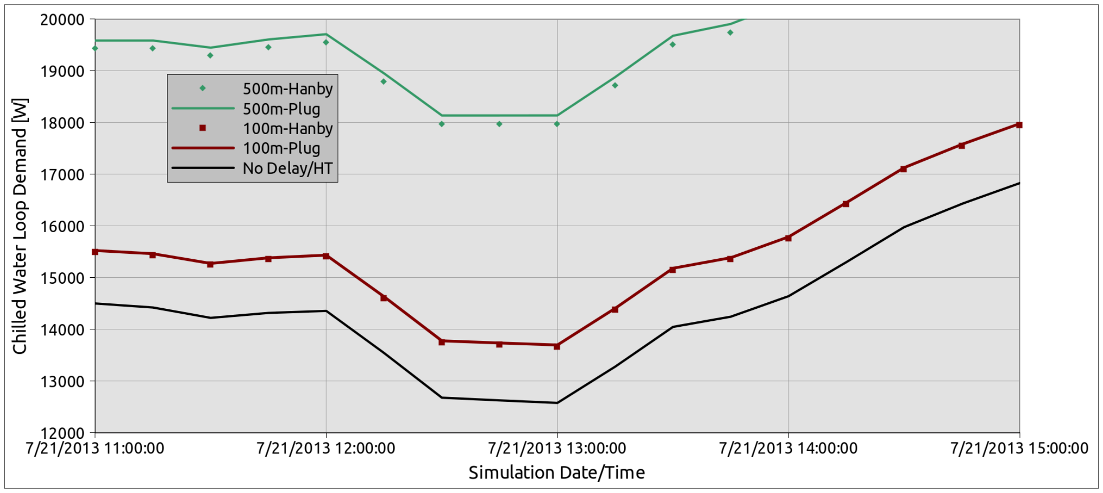


Figure 3.16: Closer view of chilled water loop demand profiles for a single simulation day under different transport modeling configurations

Instead of the delay causing an effect on the loop demand, the heat transfer to the loop appears to be the dominating effect on the loop. This indicates that further work is required to isolate the effects of the delay without having a heat transfer impact.

3.7.3 Case 2: Isolated Delay Model

The previous study showed that by adding a pipe heat transfer/delay component onto a chilled water system, the demand is not drastically impacted by a lag effect, but more prominently affected by the heat transfer added/rejected to the loop. To remove this effect, the model was modified further to approximate an adiabatic situation. The purpose is to capture a case where the heat transfer from the pipe may be minimal, whereas a long run of pipe could cause a significant delay. The approximation is employed by overriding the boundary temperature on the pipe exterior, using the current fluid entering temperature value:

$$T_{air} = T_{f,i} \tag{3.51}$$

If the system were steady state, this would result in zero heat transfer between the fluid and the environment. However, in this transient model, heat transfer can exist due to heat storage in the pipe wall itself. If the fluid entering temperature varies significantly, the transient storage in the pipe will be more prevalent, but expected to be much less than in the previous case.

In addition to the source and input changes, the loop demand reporting is replaced with the temperature difference across the pipe component:

$$\Delta T = T_{fluid,out} - T_{fluid,in} \tag{3.52}$$

In this way, the effects of other components on the loop are removed. The temperature response is important in cases where the loop response is dependent on the temperature distribution. Consider an economizer application, where the heat transfer through the heat exchanger is passive, and dependent only on the entering fluid temperatures. The amount of heat transfer available to reject through economizing will be heavily dependent on the temperature response of the delay component.

The same input file is utilized as in the previous case, and the same configurations are simulated, however there is no representative temperature difference for a base case, since the base case has no component. The results of this are shown in Figure 3.17.

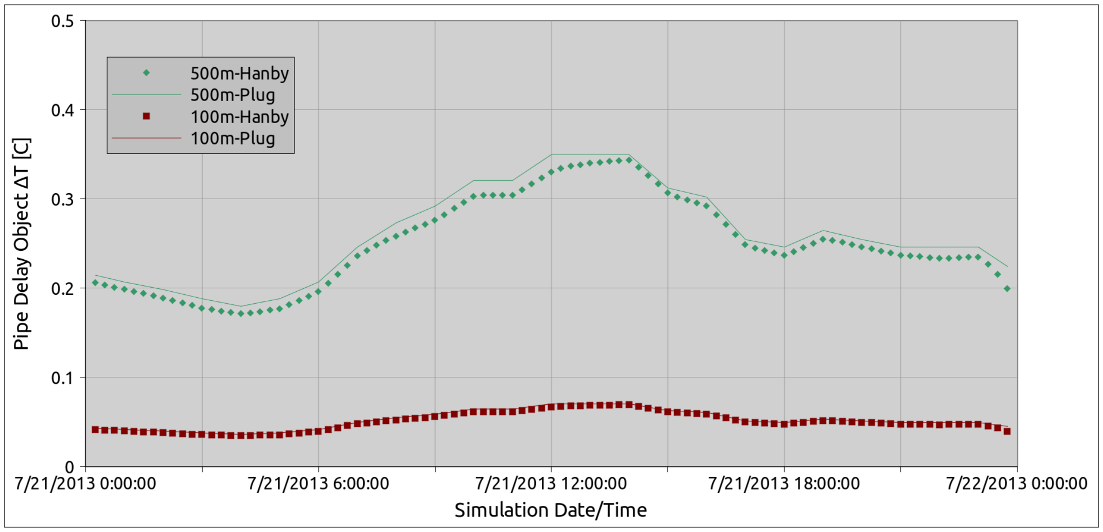


Figure 3.17: Temperature difference across delay component with boundary heat transfer minimized.

In Figure 3.17, the temperature difference is a maximum for the 500 m configurations at about 0.35 °C. For the 100 m configurations, the temperature difference is smaller, less than 0.1 °C. For reference purposes, the operating flow rate on this loop was approximately 3.29 kg/s.

Continuing the application of a water-side economizer, adding this delay with a minimized boundary heat transfer effect would result in a difference in approach temperature of between 0.05 °C and 0.35 °C during this day, depending on the configuration. The flow and demand on this loop result in a typical operating ΔT across supply equipment of 2 °C. As such, the effect of the delay is a significant portion.

The effect of a delay could be blurred on loops where the delay object is upstream of ideal, controlled, or non-temperature dependent components. These components could provide an outlet temperature regardless of the delay effect. As such, the delay effect would resolve to a simple heat transfer interaction with the loop. However, if the delay is added directly upstream of temperature dependent components, the effects will be much more prevalent. In the previous case, the loop demand was not affected significantly because of the idealization of component models, and their lack of temperature dependence.

The selection of a delay model itself appears to be much less important. Just as in the previous case, the results in Figure 3.17 show that there is a difference between models, however it is not significant. Both the well-mixed (Hanby) and plug flow models are likely to provide a suitable representation that is within the uncertainty present in the system measurements.

3.8 Conclusions

An experimental data set was created for investigating the detailed phenomena of transport delay, and used in a set of demonstrations of a model testbed. The testbed was able to bracket the experimental data using bounding modeling approaches which are fundamentally based on distinct transport phenomena described early in the discussion. The model was able to match the experimental data much better with a blend of results from different models, though this is likely to be highly dependent on specific configurations, so broad guidance is not provided by this work, only a demonstration. In addition to these bracketing cases, a third approach was expected to provide another option (Hanby et al., 2002), however it was determined that this model was simply a manifestation of the already-in-use well-mixed model, so further work directly with this approach was not necessary. Both model approaches were implemented as a component model in the central plant simulation engine inside EnergyPlus. Two simulation studies indicate that the effect of delay is important in loops with temperature dependent performance, but the effect on energy usage can be blurred by idealized component models.

CHAPTER 4

Efficient Horizontal Ground Heat Exchanger Simulation with Zone Heat Balance Integration

Abstract

For horizontal heat exchangers buried near a building slab or basement, interaction between the heat exchanger and the zone can be significant. For heat exchangers with multiple pipes in close proximity to each other, the thermal interference effects can also be significant. Previous simulation methodologies including line source and numerical solutions do not model these phenomena in a general manner. Furthermore, previous modeling approaches lack integration with other simulation domains including zone heat balance calculations and fluid loop solvers.

A numerical model for horizontal ground heat exchanger applications is presented, featuring a computationally efficient mesh and flexible heat exchanger tube placement. The model integrates the ground with zone heat balance calculations and hydronic system simulation through boundary conditions. The model is implemented within a whole building energy simulation program. Thermal interaction effects between heat exchanger pipes are captured including circuiting effects of the fluid flow direction in individual pipes.

The model is validated using experimental data taken at a test facility currently researching foundation heat exchangers. Undisturbed ground temperature data is used to estimate ground and boundary properties. Fluid temperature and zone heat transfer validation is then performed with the estimated parameters. With a full system simulation in place, the model predicts heat pump entering fluid temperature with mean bias error of 1.3 °C (2.3 °F) and basement wall heat flux with mean bias error of 1.1 W/m^2 (0.35 btu/h ft^2). This accuracy is achieved with a coarse grid that ensures a small computational footprint suitable for implementation in a whole building energy simulation program.

This work has been accepted for publication as a journal article (Lee et al., 2013).

4.1 Background

As time passes, and concerns about energy usage become more prevalent, increased demand is imposed on the building design industry to achieve improved efficiency and lower energy footprints. Stricter energy standards are increasing the requirements of building modeling as a means to evaluate designs and energy conservation measures. Guidelines have been provided by Stocki et al. (2007) relating to proper model parameters and assumptions, though no details were provided for handling ground heat transfer effects. Thomas and Rees (2009) showed that the earth heat transfer through building floors can be significant, while Adjali et al. (2004) and Andolsun et al. (2010) stated that there is much uncertainty in ground heat transfer prediction based upon modeling approach and inputs. Ihm and Krarti (2004) developed a detailed foundation heat transfer model and implemented it in EnergyPlus (Crawley et al., 2001), improving the heat transfer modeling capabilities for ground-coupled zones.

Smaller energy footprints of highly efficient buildings have opened the door for new heat exchanger configurations, including placement in the near-field of a building foundation or basement (Cullin et al., 2012; Xing et al., 2011; Den Braven and Nielsen, 1998). These foundation heat exchangers have been modeled by Xing et al. (2011). The current work builds on that with additional simulation capabilities:

- Direct coupling to a zone heat balance within a whole building energy simulation environment
- Improved flexibility for pipe placement within the calculation domain
- Capturing enhanced effects including axial temperature distributions and circuiting/flow direction effects with multiple pipes
- Improved computational efficiency using an intelligent mesh scheme

The model is applicable to foundation heat exchangers, horizontal heat exchangers and district heating or cooling systems. The grid generation techniques used make this model suitable for simulation of long piping systems, as the computational mesh is refined in areas where the thermal interaction is highest, and the model is proven to provide accuracy with a highly coarse grid.

In addition, the model can be applied to niche configurations, including modeling the supply water pipe from a utility junction to a building, modeling the horizontal legs between vertical boreholes in a ground heat exchanger field, and multiple pipe configurations with heating, cooling, or neutral pipes running in proximity. The model fully accounts for the effects of circuiting and flow direction and is suitable for implementation in a whole building energy analysis program.

4.1.1 Preliminary Modeling Discussion

Existing horizontal ground heat exchanger models have three shortcomings when applied to novel configurations in whole building energy simulation. The first is the lack of generality. Approaches using a line source allow generalized pipe placement (multiple pipes with superposition), but are limited in integration capabilities. Building simulation fluid loop solvers using a flow-wise component-by-component simulation order are designed for component models that input entering fluid conditions and return fluid exiting conditions. The line source model is driven instead by the line source intensity, or the heat rejection rate of the source. Several studies (Ingersoll and Plass, 1948; Den Braven and Nielsen, 1998; Chengju et al., 2012) utilized line source theory to simulate buried pipes and heat exchangers. Other studies (Ngo and Lai, 2005; Sadegh et al., 1987) used a simplified representation of the fluid as a boundary condition to the ground domain.

The second shortcoming is the lack of coupling to entire fluid loop simulation engines. This is enabled by simulating the transit of fluid through the model, from an inlet to an outlet. This allows coupling the model to a fluid loop to evaluate the energy usage or dynamic response of an entire system. Numerous studies on buried pipes or heat exchangers (Bau and Sadhai, 1982; Bronfenbrener and Korin, 1999; Chung et al., 1999; Esen et al., 2007; Said et al., 2009) modeled the fluid without capturing the transit effects, limiting the possibility of performing whole system evaluation. Yavuzturk and Spitler (1999) described a vertical ground heat exchanger model that relies on response factors to calculate the fluid response through the heat exchanger, and includes the fluid transit effects. Tobias (1973) used an approximation of the fluid response in the system to allow fluid transit to be captured in simplified models. Mei (1988) and Piechowski (1999) utilized specialized coordinate systems to capture the fluid transit with either one or two pipes in the domain. The dual coordinate system approach for embedding pipes in the domain by Piechowski (1999) is a suitable starting point for developing a generalized horizontal ground heat exchanger model, because the grid is refined in the near pipe region, without the need for a complicated or highly dense coordinate system.

The third shortcoming in existing models is the lack of integration between the ground and zones. In whole building energy simulation programs, ground heat transfer models must be integrated with the zone heat balance calculations to account for dynamic thermal feedback. Binks (2011) noted the importance of accurate ground temperature prediction for building simulation, though zone heat balance simulation models generally use a simplified representation of the ground, utilizing a direct boundary condition on the bottom of the ground-coupled floor surface. Cullin et al. (2012) utilized an iterative approach to couple separate zone and ground heat exchanger models when simulating foundation heat exchangers. This development path resulted from the idea that thermal ground interaction was secondary to other heat transmission in the zone (Claesson and Hagentoft, 1991). The interactions between the zone and a ground heat exchanger are secondary in traditional heat exchanger configurations, as there is sufficient distance between the two to decouple them. For low energy applications where these heat flows are more dominant, this assumption can break down.

Coupling the three domains: ground, zone, and fluid, is a major contribution of the current model. In addition to this integration, the effects of pipe placement, flow direction, thermal interference between pipes, and circuiting are captured by the model. Simulation mechanics and assumptions vary between simulation programs, however it is common for the zone heat balance to be a quasi-steady-state solution. Pumping and piping simulation is often similar, with steady state mechanics utilized over a single step in time. Coupling these quasi-steady simulation mechanics to a transient ground simulation model requires special treatment of the various domain hooks. This is further complicated if the simulation domains operate at independent time step levels, such as with the whole building energy simulation tool EnergyPlus (Crawley et al., 2001). Coupling the different simulation mechanics and independent time integration steps is addressed by the current model which improves the feedback between simulation systems and improves accuracy of the whole building energy simulation environment.

4.2 Methodology

The physics of the ground heat exchanger model consist of thermal interaction between a fluid being transported through the domain, the transient ground mass, and the various boundary conditions including the ground surface, zone heat balance, and far-field. The physical domain can contain multiple pipes located near a basement zone, possibly in the excavation area of the ground. By simplifying the geometry into a Cartesian simulation domain and assuming a far-field boundary distance, the corresponding simulation domain is shown in Figure 4.1. In this figure, the domain cross section contains a basement region, and as an example, there are five tubes placed in the domain. The domain consists of a series of these two-dimensional cross

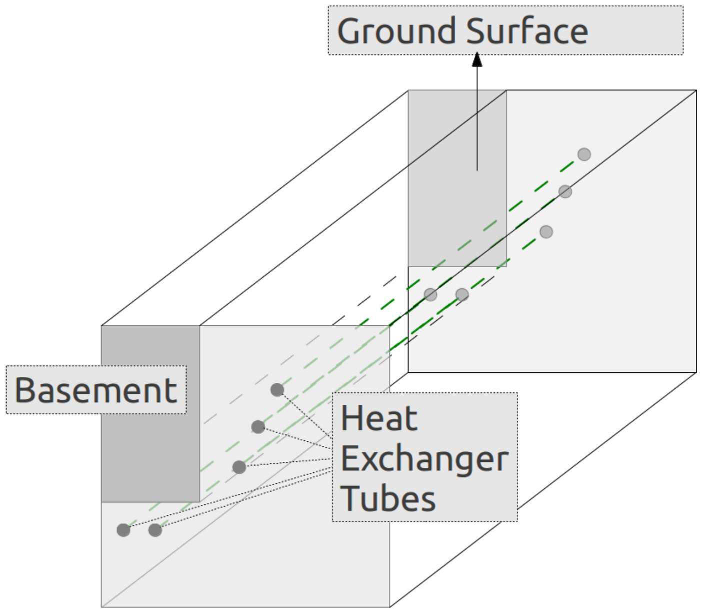


Figure 4.1: One possible simulation domain that includes heat exchanger pipes and a basement zone

sections extruded uniformly in the axial pipe direction. Thus all pipes and any other objects in the domain are parallel with uniform geometry throughout the axial length. This assumes that any zone interaction exists over the entire length of the domain. When basement walls only exist for a portion of the domain length, multiple domains are implemented, of which some will include basement interaction and some will not. Based upon the required detail, careful circuiting of the fluid between the domains can be implemented to ensure the fluid path is exactly as in the real system. For the case of foundation heat exchangers, the tubes may “wrap” around multiple corners of the basement in reality. The model assumptions do not allow this to be applied directly. Instead, the physical domain must be simplified with an effective overall length to capture the corner effects.

As shown in Figure 4.1, the basement region is a rectangular section that is “cut-away” from the ground domain. The size of the cutaway is variable and is selected to fit particular applications. In cases where the floor heat transfer is significant, such as if pipes are placed underneath the floor, the entire basement floor may be modeled. Whereas if this is less significant, a smaller representation of the basement floor can be utilized and will represent the entire floor. Of course if there is no basement in the pipe region, then this will not be present in the domain, and the farfield will be applied at the Cartesian domain boundary.

4.2.1 Simulation Domain

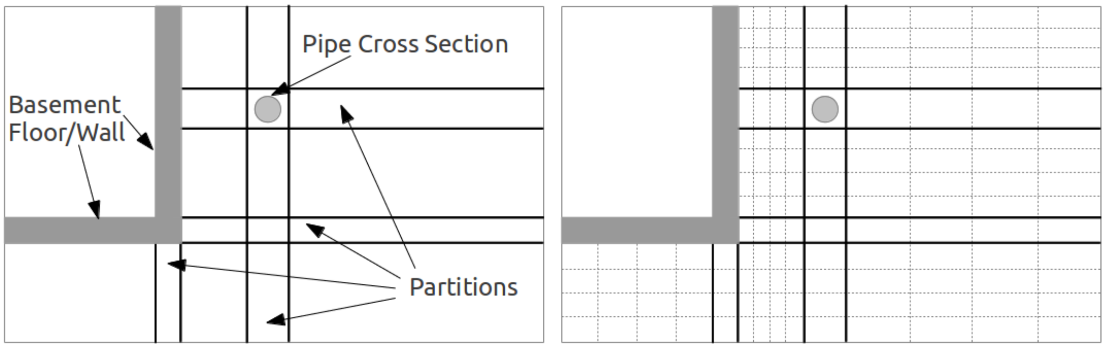
The simulation domain consists of the ground, plus the integration with the zone and piping systems, along with other boundary conditions. Groundwater movement is not included, but the effects of stagnant moisture content in the soil, including freezing, are simulated. Moisture transport effects are excluded because parameters required for groundwater flow models are only known under specialized conditions. Raymond et al. (2011) demonstrated (through validation of a numerical model using data from an experimental test site (Austin et al., 2000)) that for a significant range of groundwater flow conditions, the effects on a thermal response test are negligible. It is assumed that the inclusion of stationary moisture content can provide sufficient accuracy. The freezing is simulated using an effective specific heat over a small temperature range near the freezing point. The total energy within this range is equivalent to the latent heat of melting. This method is described by Lamberg et al. (2004).

The heat transfer in the ground is governed by a transient energy balance:

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T \tag{4.1}$$

This equation is applied to a mesh created in the domain. The coordinate system is Cartesian, suitable for the rectangular domain (Figure 4.1). Since the domain will contain objects besides just the ground, the mesh is created using a partition approach. Vertical and horizontal partitions are aligned in the domain at the location of each pipe or domain object. A single pipe in the domain, along with the basement surfaces, results in two partitions in each of the x and y directions, as shown in

Figure 4.2a. The partition is a finite size, large enough to contain the pipe or basement surface. Vertical partitions become a single cell wide, and horizontal partitions become a single cell tall as part of the overall mesh.



(a) Partitions placed in the domain (b) Spaces between partitions meshed

Figure 4.2: Domain visualization for the partition based mesh development procedure

The regions between the domain partitions are then meshed, as shown in Figure 4.2b. The mesh may be uniform throughout the region or utilize a symmetric geometric series expansion to define the cell distribution. A uniform mesh distributes the cells evenly. The geometric distribution is calculated based on the number of cells and an expansion coefficient ζ. The geometric distribution is symmetric, thus one side of the region is meshed, then mirrored to the other half. The width of each cell is calculated as:

$$\Delta x_1 = \frac{\Delta x_{region}}{2} \left[\sum_{j=0}^{N_{cells}/2} \zeta^j \right]^{-1} \tag{4.2}$$

$$\Delta x_i = \Delta x_1 \zeta^i \tag{4.3}$$

Once complete, the domain may be meshed as in Figure 4.2b. The number of cells is the same in each mesh region between objects. As partitions get closer, the grid then becomes refined, which is beneficial as these areas would be expected to have the highest level of thermal activity. This refinement is enhanced if the geometric mesh distribution is utilized. The domain is then extruded in the pipe axial direction to provide three dimensional cells.

4.2.2 Coupling: Pipe & Ground

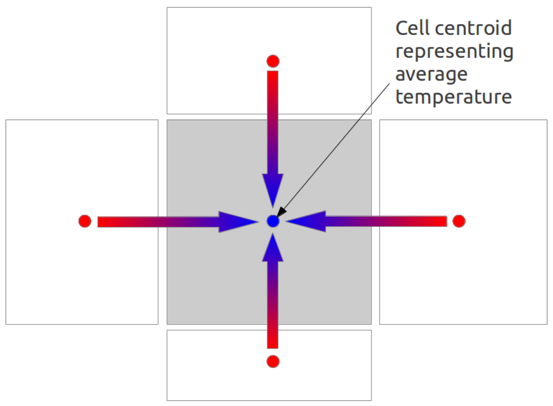
The fluid passes through the three-dimensional domain inside each pipe segment. The flow direction is defined per segment, and the same flow can pass through multiple pipes. This allows a single circuit to have multiple passes within the domain, capturing the effects of flow direction. Multiple circuits can then be placed in the same domain to allow multiple fluid inlets and outlets. The transfer from one segment to another is idealized, the effects of a u-bend at the end of the domain are not simulated. Instead, the fluid information is immediately transferred from one outlet to the next segment inlet.

Figure 4.3 shows different approaches to simulate the pipe within the Cartesian grid. Utilizing a single temperature for the entire cell, which is an average of the contents in the cell, is shown in Figure 4.3a. With this method, it is difficult to capture the fluid-soil interaction, as the effects are lumped.

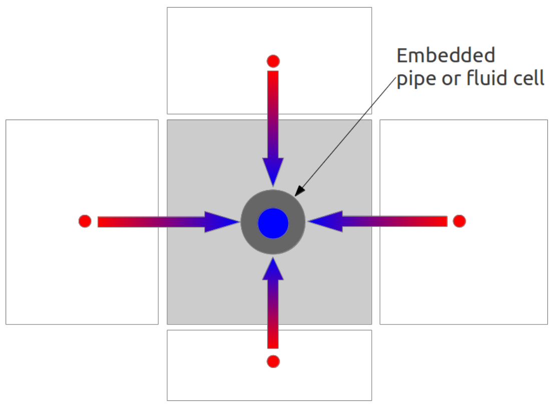
An additional level of detail is shown in Figure 4.3b, in which the fluid and pipe are explicitly modeled. This is a suitable approach, however, the mesh near the pipe is as coarse as the surrounding Cartesian system. Since this area contains the highest thermal activity, this region warrants additional refinement.

Utilizing a radial coordinate system embedded within a Cartesian cell was proposed by Piechowski (1996). Figure 4.3c shows a full radial coordinate system placed within the Cartesian cell. Note that this results in an interface cell which exists at the four corners of the Cartesian cell boundaries. The surrounding Cartesian system interacts with this interface instead of directly with the embedded pipe cell. The radial system is then utilized to simulate the near-pipe region, and inherently provides a refined mesh in this region.

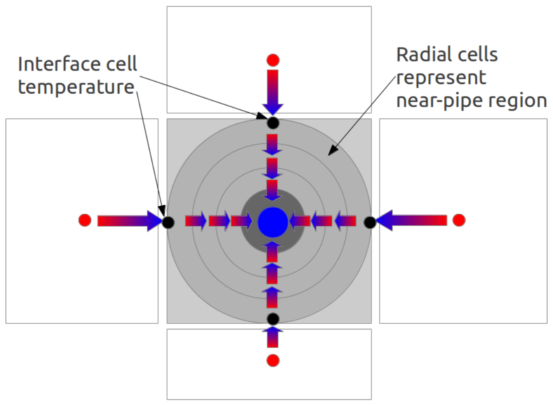
The radial system could remain with interactions in each of the Cartesian direc-



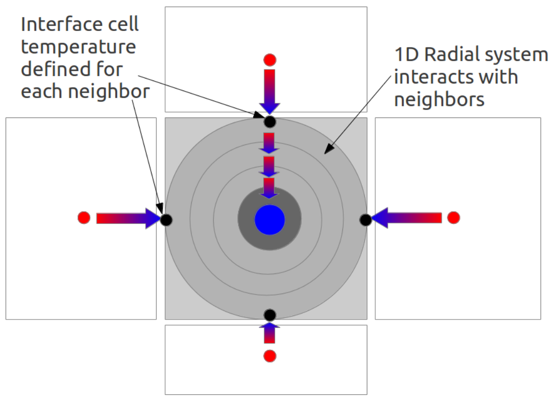
(a) Using a single average cell temperature



(b) Simplified representation of pipe or fluid



(c) Embedded radial coordinate system



(d) Embedded axisymmetric radial coordinate system

Figure 4.3: Approaches to simulate pipe cell effects within a Cartesian coordinate system domain

tions. However, the fluid cell, which is developed in a following section (4.2.3) uses a uniform condition at each axial cross section. Thus, any angular variation will ultimately not be recognized by the underlying fluid. To improve computational efficiency, the calculations are reduced to an axisymmetric radial system as shown in Figure 4.3d. All four Cartesian neighbors interact with the interface cell, along with the single radial direction. The current model builds upon the original dual coordinate system approach by Piechowski (1996) with fully generalized pipe placement in the domain.

The thermal interchange between the coordinate systems is governed by the following energy balance:

$$\rho V C_p \frac{\partial T}{\partial t} = \sum \dot{q}_{in,Cartesian} + \dot{q}_{in,radial} \tag{4.4}$$

For the Cartesian heat transfer calculations, the thermal distance is the distance from the centroid of the neighbor Cartesian cell to the corresponding interface thermal node, as represented by the arrow mapping from a Cartesian cell into the coordinate system interface in Figure 4.3d.

$$R_{Cartesian \mapsto interface} = \frac{\Delta y_i}{2 k_{i,j} \Delta x} \tag{4.5}$$

For the radial system a standard radial resistance is applied:

$$R_{Radial \mapsto interface} = \frac{\ln\left(r_o / r_c\right)}{2 \pi k} \tag{4.6}$$

From the interface inward radially to the pipe wall, the heat transfer is modeled using a transient radial formulation. At the pipe wall to fluid interface, the approach requires special treatment, as described next.

4.2.3 Coupling: Pipe & Fluid

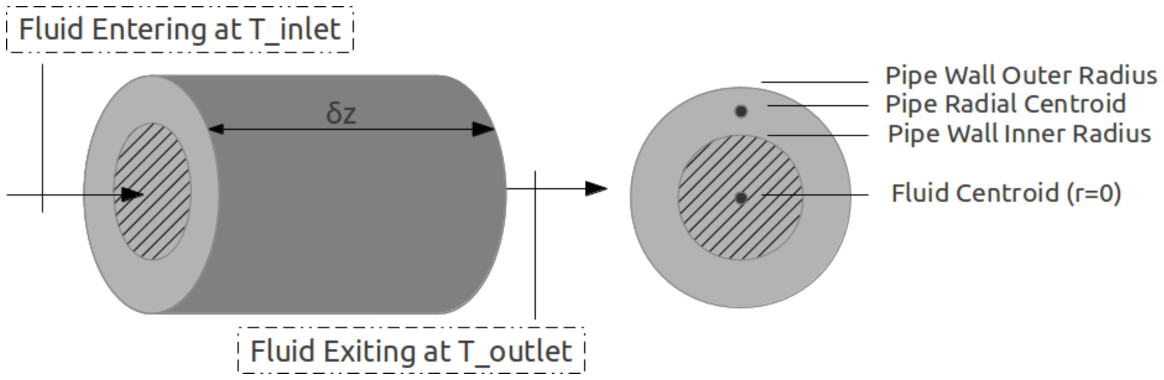


Figure 4.4: 2D and 3D representations of the fluid cell, with radial coordinate system nomenclature

Figure 4.4 shows a number of features related to the fluid cell geometry. The fluid cell is a cylindrical finite volume cell with a representative temperature located at the center of the flow. The fluid inlet at any cell is a well-formed boundary condition of temperature and mass flow rate. The pipe wall is a radial finite volume cell with a representative temperature located for thermal network calculations at the radial centroid (see Figure 4.4). The fluid and pipe cells are coupled via the heat convection at the pipe inner surface. During a given iteration, the pipe wall has a single, uniform temperature. By assuming the entering fluid mixes with the fluid currently existing in the cell, the governing equation is the following energy balance:

$$mC_p \frac{\partial T_f}{\partial t} = \dot{m} C_p \left(T_{in} - T_f \right) + UA \left(T_{pipe} - T_f \right) \tag{4.7}$$

The accuracy provided by this equation will depend on the domain configuration, especially the axial length of each pipe segment. The actual phenomena occurring within the pipe at a given point in the system includes mechanical mixing, thermal diffusion, boundary heat transfer, entry-length effects and varying pipe geometry and piping connections. Assuming the flow is generally turbulent, this equation which assumes mixing provides the accuracy required for whole building simulation applications.

The surface conductance U is calculated as the series radial resistance from the fluid to the pipe wall radial centroid, thus including convection and conduction. For turbulent flow, the convection coefficient is calculated based on current fluid conditions using a well-accepted form of the Nusselt correlation by Dittus and Boelter (1930):

$$Nu_D = 0.023 Re_D^{0.8} Pr^n \;\; [Re_D \geq 2300] \tag{4.8}$$

The exponent n is set to 0.4 when the fluid is being heated and 0.3 when the fluid is being cooled. This equation is an explicit expression for Nusselt number that assumes a uniform set of properties over the fluid. In heat exchanger applications of this model, the temperature variation is low enough to allow the use of such an equation. Through the course of a simulation, the thermal properties of the fluid are updated, however for a single iteration the thermal properties of the fluid are fixed.

Although laminar flow conditions are not expected in heat exchanger applications, this model uses a constant value of Nusselt number as shown in Equation (4.9) for low Reynolds Number conditions. The constant value is an analytic solution assuming fully developed, steady, one-dimensional flow in a circular tube with a constant surface temperature.

$$Nu_D = 3.66 \;\; [Re_D < 2300] \tag{4.9}$$

Axial temperature variation in the soil, pipe and fluid are predicted in this model. The use of this constant surface temperature result for Nusselt relies on an assumption that the radial temperature distribution provides a more significant effect on the heat transfer phenomena. Other approximations could be utilized (for example assuming a constant wall heat flux, $Nu_D = 4.36$), but are not expected to provide any additional accuracy. The current set of applications for this model do not warrant a more detailed approach.

Under zero-flow conditions, a prescribed convection coefficient simulates the free convection heat transfer in the pipe.

The fluid cells are simulated flow-wise from the circuit inlet and downstream to the circuit outlet. Circuit inlet conditions are well-formed from other components in the fluid loop simulation system. The flow-wise simulation captures directional flow circuiting effects. The integration between this fluid circuit and the entire fluid loop simulation model is described next.

4.2.4 Integration: Fluid Loop Simulation

The fluid heat transfer within the pipe is governed by the equations in the previous section. The fluid circuit inlets and outlets of this model are then connected to a fluid loop simulation within the whole building energy analysis program. At each iteration, the fluid circuit inlet provides a momentum and energy boundary condition specified by temperature and flow rate conditions. The fluid outlet condition is governed by continuity and the energy balance on the outlet cell of the circuit (equation (4.7)). The loop simulation is quasi-steady state operating at a variable time step. The time-stepping solver can also back-step and repeat a time step over as necessary to achieve system convergence. The ground domain is fully dynamic, thus the coupling between the two domains acts as an interface between time stepping paradigms. The fluid and near pipe region must tentatively step in time at each simulation call, capturing the transient response from the fluid passing through the domain. However, since the loop solver is quasi-steady-state, the entering fluid temperature in the next iteration may be drastically different, and the tentative results must be overridden until convergence is attained.

In terms of this current research project, this fluid loop simulation model was developed as a major research project. This is described fully in chapter 2.

4.2.5 Integration: Zone Heat Balance

Integrating the zone heat balance with the ground simulation, and therefore with the fluid in the heat exchanger captures the thermal feedback between the two systems. The zone heat balance is governed by equation (4.10), which is a transient energy balance of thermal phenomena in a zone. The first term on the right hand side is the sum of internal gains in the space (people, equipment, lights). The second term represents the infiltration gain on the space. The third term represents the convective heat transfer from each surface in the space to the air. The final term represents the energy provided by system conditioning equipment:

$$m_a c_{p,a} \frac{\partial T_a}{\partial t} = \sum_{i=1}^{N} \dot{q}_{int} + \dot{q}_{inf} + \sum_{i=1}^{N_{surf}} \dot{q}_{conv} + \dot{q}_{sys} \tag{4.10}$$

The zone air is then connected thermally with the zone surfaces via the convective heat transfer rate, governed by Newton's law of cooling:

$$\dot{q}_{conv} = hA\left(T_{surface} - T_a\right) \tag{4.11}$$

The convection coefficient, h, is a function of several variables, depending on the model used for the zone air conditioning. Generally it will be a function of diffuser type and location, surface orientation and overall zone air flow rate.

The heat transfer through the surface is transient conduction, which is typically modeled using a conduction transfer function method or a finite difference algorithm. Conduction transfer functions are used widely in whole building energy simulation due to the lightweight computational burden. Response factors are generated one time for each construction, and these are then used in a time series calculation to determine the response of the surface. For calculating the heat flux on the inside of a building surface, using the temperature and heat flux histories of the wall, the response factor expression takes the following form:

$$\begin{aligned} q''_{in}(t) = &-\sum_{j=0}^{N_Z} Z_j T_{in,t-j\Delta t} \\ &-\sum_{j=0}^{N_Y} Y_j T_{out,t-j\Delta t} + \sum_{j=1}^{N_\Phi} \Phi_j q''_{in,t-j\Delta t} \end{aligned} \tag{4.12}$$

In equation (4.12), the terms Z, Y, and Φ represent the conduction transfer coefficients. While conduction transfer functions are rapid and convenient for whole building energy simulation, they cannot be used directly in the simulation of surfaces with variable thermal properties. Barbour and Hittle (2006) pre-calculated extra sets of conduction transfer functions to handle variable properties. The number of extra sets ranged from 3 to 599,999 to achieve proper accuracy. The computational cost associated with the higher sets of transfer functions limits the application of such methods in whole building energy simulation.

The exterior of the zone surface is then coupled to the ground domain. This is performed using a convective boundary with a large value of surface conductance. This essentially becomes a temperature boundary on the surface. The ground domain supplies the surface heat balance with an average temperature for transient surface conduction calculations, and in return the surface heat balance supplies the ground domain with an average heat flux. The ground domain uses this heat flux as the boundary for cells adjacent to the surface.

Integration of the ground domain and the zone heat balance occurs at the surface exterior. Other possibilities exist, such as including the zone surface directly in the ground model domain. The coupling to the zone air heat balance would then occur at the convective boundary between the zone air and the wall interior surface. Allowing the surface heat balance manager to simulate the wall itself allows the wall solution type to be separate from the ground model. The wall may then be simulated using transfer functions or a finite difference approach, and could contain variable properties or other specialized features.

The ground domain is simulated at the time step of the air and fluid system simulation. The zone and surface heat balance equations occur at a different time step. Thus, the boundary conditions imposed on the ground domain are constant during all ground time steps until the next surface time step. When the surface begins a new time step, the aggregated energy added by the ground domain over the previous time step is used as the boundary. Since the integration occurs at the exterior surface of the domain, and the zone typically runs at time steps less than one hour, the lag will be insignificant over the course of a long-term simulation. Very light-weight surfaces may be more prone to inaccuracy with this assumption, however this effect is further dampened if the zone is well-controlled, such that the inside temperature is nearly constant.

4.2.6 Ground Domain Boundary Conditions

A Dirichlet condition ($T = T\left(z, t\right)$) is applied on the ground domain at the far-field faces. In this model, cell centroids are not aligned directly at the outer boundary surface, so the far-field temperature is applied to the outer surface of the cell. An energy balance is evaluated on the cell to determine a centroid cell temperature. The boundary temperature is calculated at a given time and depth using a standard expression introduced by Kusuda and Achenbach (1965):

$$T(z,t) = \overline{T}_s - \overline{\Delta T}_s \times e^{-z\sqrt{\frac{\pi}{\alpha\tau}}} \times \cos\left(\frac{2\pi t}{\tau} - z\sqrt{\frac{\pi}{\alpha\tau}} - \theta\right) \tag{4.13}$$

Three parameters, $\left\{\overline{T}_s, \overline{\Delta T}_s, \theta\right\}$, must be estimated from knowledge of the ground temperature variation, either approximated from weather or location data, or generated from experimental ground temperature data.

The ground surface energy balance includes convective heat transfer as well as radiation and evapotranspiration on the exterior surface, with conduction to the interior of the domain. Evapotranspiration is modeled using the approach presented by Allen et al. (1998), which governs the rate of evapotranspiration according to:

$$h_{fg}E = \frac{\delta\left(G_r - G_s\right) + \frac{\rho_a c_{p,a} e'}{r_a}}{\delta + \gamma\left(1 + \frac{R_s}{R_a}\right)} \tag{4.14}$$

4.2.7 Solution Algorithm

The ground domain is solved with an inherently stable implicit numerical formulation to ensure robustness within the variable time step environment. The system of equations is solved via iteration. Initialization of the domain is performed using a thermal gradient in the domain according to the far-field boundary specification. Convergence of the iteration is determined by a specified maximum absolute temperature change in the domain.

The integration between simulation domains adds complexity to the solution scheme for the model. As already mentioned, the zone heat balance occurs at a time step larger than the ground model, so that the effects are lagged between the two domains. In addition, the fluid loop solver is an iterative quasi-steady solution that can both vary the time step and back step within the main iteration loop. It is expected that the ground domain will respond fastest in the near pipe region, where temperatures could rapidly change based on loop conditions. Because of this, the ground domain thermal network is updated at variable time steps, aiding in a lightweight computational footprint by not simulating the ground at each iteration. The overall time step operation and model calling points are shown in Figure 4.5. The ground domain is updated on the first system time step, while the fluid circuit is shown to be embedded inside the system time integration loop. Figure 4.5 also shows that the surface temperatures are updated at a higher level, resulting in a lag of information transfer between the domains.

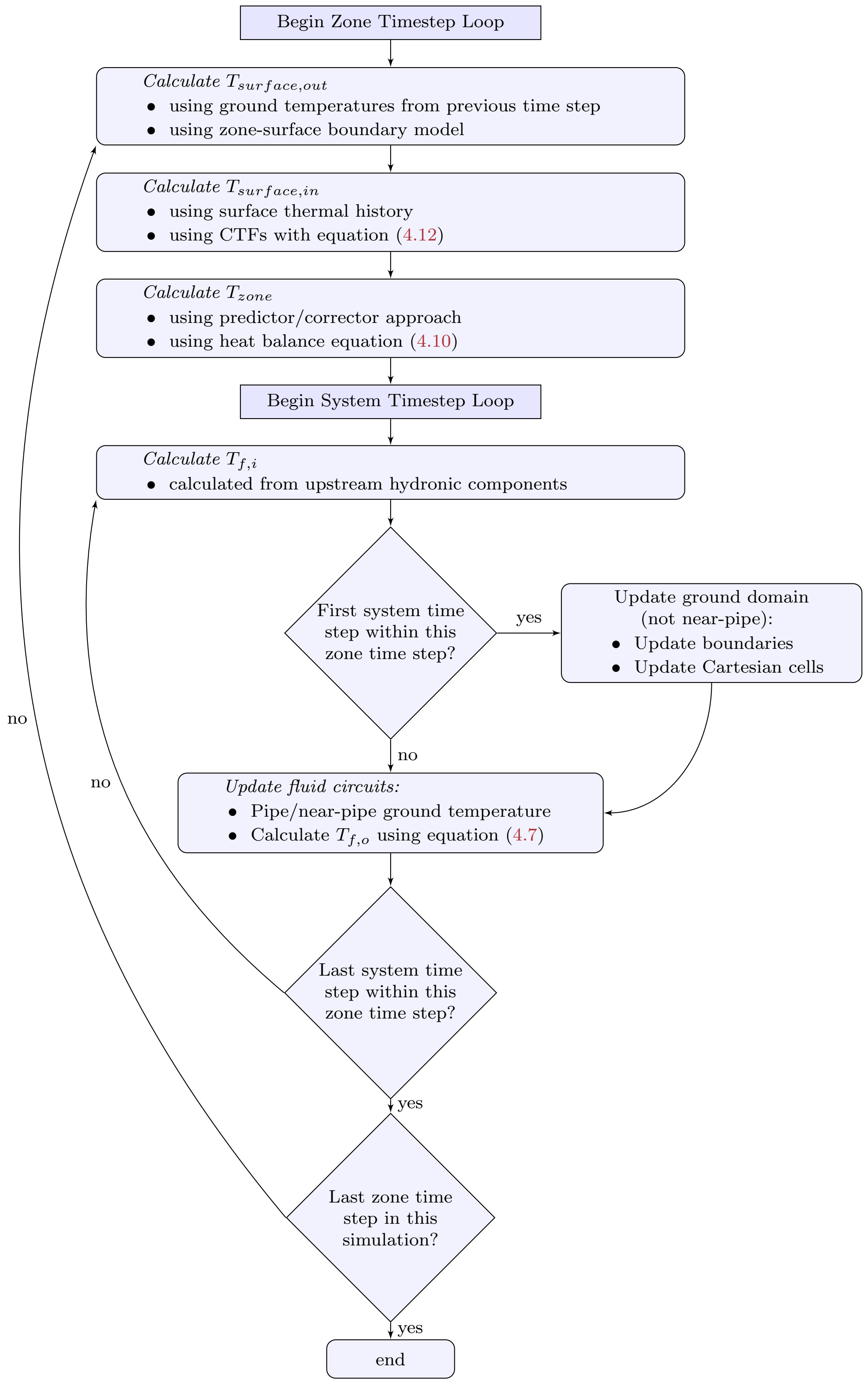

Begin Zone Timestep Loop
Calculate $T_{surface,out}$
• using ground temperatures from previous time step
• using zone-surface boundary model
Calculate $T_{surface,in}$
• using surface thermal history
• using CTFs with equation (4.12)
Calculate T_{zone}
• using predictor/corrector approach
• using heat balance equation (4.10)
Begin System Timestep Loop
Calculate $T_{f,i}$
• calculated from upstream hydronic components
First system time step within this zone time step?
yes
Update ground domain (not near-pipe):
• Update boundaries
• Update Cartesian cells
no
Update fluid circuits:
• Pipe/near-pipe ground temperature
• Calculate $T_{f,o}$ using equation (4.7)
Last system time step within this zone time step?
no
yes
Last zone time step in this simulation?
no
yes
end

Figure 4.5: Solution logic of integrated modeling system, showing relevant variable calculation points

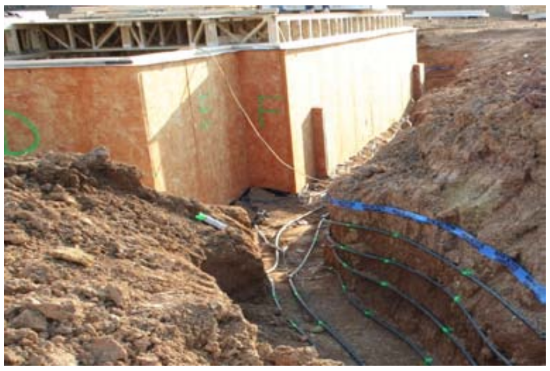
4.3 Model Evaluation

Foundation heat exchangers (Spitler et al., 2010) are a special type of ground heat exchanger placed in the excavation area of a basement. This placement results in significant thermal interaction with the zone. Multiple buried pipes can be laid in this same trench, which results in significant thermal interaction between pipes. These interactions, along with the relatively close proximity to the ground surface result in a lower heat exchanger capacity per length compared to vertical borehole heat exchangers, which interact with the nearly constant deep ground temperature. For traditional building design, foundation heat exchangers do not provide sufficient capacity, but low-energy designs with lower peak loads can make use of these in some climates and configurations (Cullin et al., 2012).

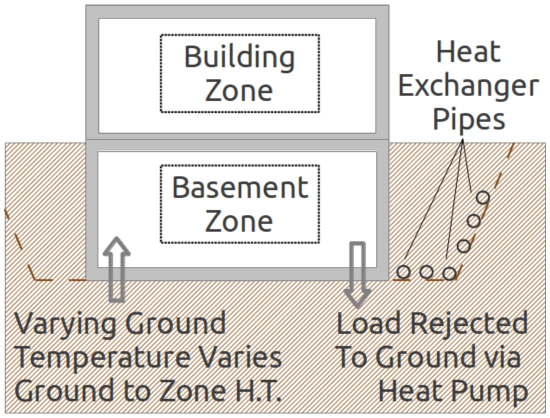
4.3.1 Experimental Facility

The foundation heat exchanger configuration provides a useful validation configuration for this modeling work as it includes high thermal activity between multiple pipes and between the ground and zone. An experimental facility in Oak Ridge, TN, USA (as described by Xing et al. (2011)) consisted of a full scale low-energy residential building with a foundation heat exchanger and a multiple pipe horizontal heat exchanger in a utility trench. A photo of this piping is shown in Figure 4.6a, with a simplified schematic of the foundation heat exchanger in Figure 4.6b. These two figures show how the tubing is laid directly into the already excavated areas. This reduces or eliminates the cost of drilling and excavation work that is done specifically for heat exchanger installation.

The fluid loop, as installed at the experimental facility, is shown in Figure 4.7a. Undisturbed ground temperature was measured away from the heat exchanger installation. This data was used to perform parameter estimation to determine model parameters. Fluid temperature was measured at multiple locations around the loop.



(a) Photo of experimental facility, from Spitler et al. (2010)


Building Zone
Basement Zone
Heat Exchanger Pipes
Varying Ground Temperature Varies Ground to Zone H.T.
Load Rejected To Ground via Heat Pump

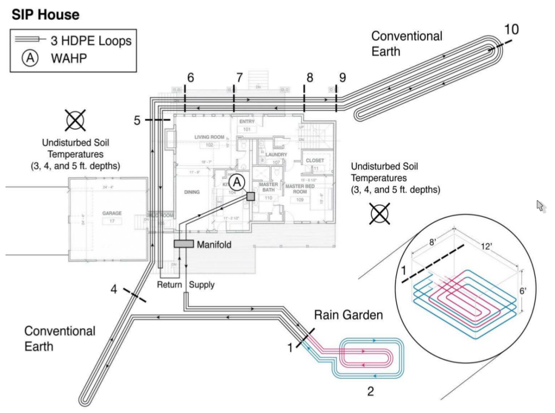
(b) Simplified schematic

Figure 4.6: Foundation heat exchanger installation and representation of thermal interaction

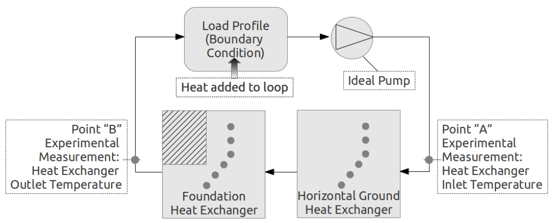
For the current validation efforts, only the loop inlet and outlet temperatures were utilized, for both component-model and system-level validation studies. Heat flux measurements were made along the basement wall at multiple depths, characterizing the effects of the heat exchanger on the zone. This heat flux data was used in validating the integration of the ground model with the zone heat balance.

4.3.2 EnergyPlus Model

The model was implemented in the whole building energy simulation software EnergyPlus (Crawley et al., 2001) as a new component in the central plant simulation algorithms. The fluid loop in the experimental facility (Figure 4.7a) consists of a foundation heat exchanger region near the basement as well as conventional heat exchangers, some of which pass through a rain garden area. In EnergyPlus, the system was modeled as two heat exchangers: the foundation heat exchanger, and the remainder of the system as a single horizontal heat exchanger, shown in Figure 4.7b. To complete the full system simulation in EnergyPlus, a load profile object was utilized to provide heat input to the loop, and an idealized pump was added to provide flow to the system. Experimental measurements of system flow rate were used as a boundary condition on the model.



(a) Experimental Fluid Loop, from Xing et al. (2011)


Load Profile (Boundary Condition)
Heat added to loop
Ideal Pump
Point "B" Experimental Measurement: Heat Exchanger Outlet Temperature
Point "A" Experimental Measurement: Heat Exchanger Inlet Temperature
Foundation Heat Exchanger
Horizontal Ground Heat Exchanger

(b) Simplified Line Drawing

Figure 4.7: Experimental fluid loop and the simplified representation used in validation efforts

For component-model validation, the model directly used the experimental data for entering temperature and flow rate, overriding any system effects, in order to isolate the validation to the component itself. In Figure 4.7b, point A represents the point where experimental temperature was applied. For system simulation validation, the component directly used the conditions entering from upstream components.

During these studies, the model algorithms were optimized for improved computational efficiency. Performing an annual detailed foundation heat exchanger simulation within the EnergyPlus whole building shell using a fully optimized version of the application took less than five minutes on a modern computer. This computational footprint is within acceptable levels for whole building energy simulation.

4.3.3 Numerical Considerations

A typical numerical modeling approach for ensuring grid independence consists of running with an increasingly denser mesh until a convergence criterion is achieved, usually a maximum temperature differential in the domain between passes. As the results become independent of the grid, the change in results diminishes. The independent domain is used for subsequent calculations.

A feature of the current model is the level of integration between the ground, zone and fluid systems. This integration provides the ability to use grid independence metrics beyond domain temperatures. Implemented within a whole building energy simulation environment, the effects on zone and the fluid system provides more relevant metrics. The grid independence study focuses on fluid and ground temperatures, but also includes the effects on zone loads, which directly impact energy use. The whole building simulation environment also requires a model that is computationally efficient. A typical grid independence analysis will produce a fully independent grid, but at the cost of an unusable grid configuration. This grid independence study balances computation and accuracy, with a focus on building energy use as a metric.

4.3.3.1 Preliminary Discussion

The grid independence study was performed varying the grid using three mesh density parameters, each of which has a distinct effect on the accuracy and computational burden of the domain:

- Cartesian Inter-Partition Mesh Density (X and Y directions treated equally in this study)
- Axial Mesh Count
- Radial Soil Mesh Count

Trials were made of each mesh parameter at the values: {1, 4, 7, 10} (64 total). The output metrics for this integrated model include the heat exchanger outlet temperature, spatially averaged basement wall temperature, and basement zone load. Each of these are averaged for an annual simulation to provide a single metric for the entire annual run.

XY Mesh Density The XY mesh density is used to define the number of cells between each partition or surface in the domain. A value of one means that a single Cartesian cell is placed between any two partitions, resulting in a highly coarse domain. This parameter refines the mesh near the zone surfaces.

Axial Mesh Count Axial mesh count is the number of cells along the length of the pipe segments placed in the domain. With a single cell, the effects of temperature non-linearity cannot be captured. The effects of fluid temperature variation along the pipe length is captured with a higher number of axial cells.

Radial Mesh Count The radial mesh count is the number of radial soil cells inside a Cartesian cell containing a pipe. Using a single radial cell can provide suitable accuracy because the Cartesian cell will also contain an interface cell, a pipe wall cell, and a fluid cell. Even with a single radial cell, the near-pipe region is refined relative to the Cartesian system. The addition of radial cells is expected to have minimal impact on results.

Overall Mesh Count The overall mesh count is a function of the three interacting mesh parameters. An increase in axial cell count increases the number of cells in the domain linearly, as it is adding domain cross sections. An increase in XY mesh or radial mesh count is dependent on the number of features in the domain. The interactions between each parameter are non-trivial, having effects on computation time, accuracy, and convergence.

4.3.3.2 Computation Time

The computation time results are shown in Figure 4.8 as a function of overall mesh count. As expected the computation time trend was to rise as the total number of cells increases. However, the curve is not monotonically increasing. The total cell count obscures the interactions between the mesh parameters. This is explained by example: The total cell count may increase as a combined result of increasing the radial count and reducing the axial count. The computation cost of additional radial cells is smaller than additional axial cells, thus the computation time can decrease even with an increase in overall cell count. As a reference, the total cell count for the configuration used in further experimental validation is labeled on the plot.

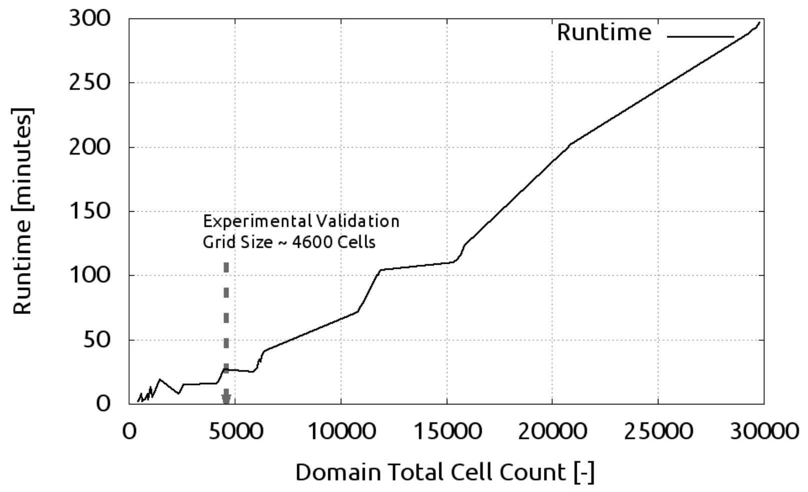
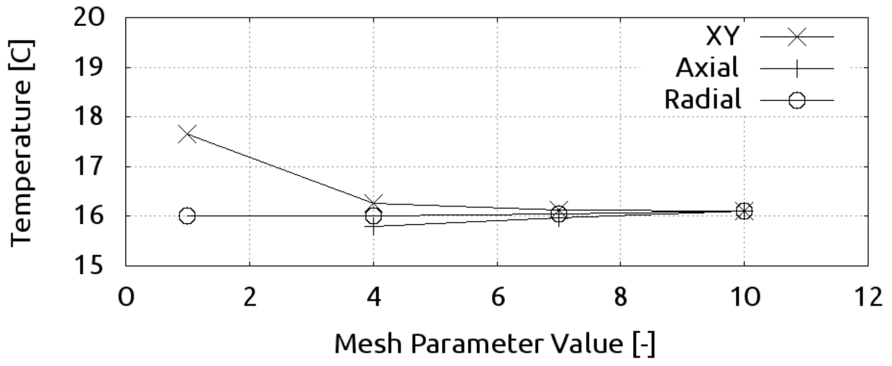


Figure 4.8: Overview of computation time increase as a function of the total number of cells in the domain

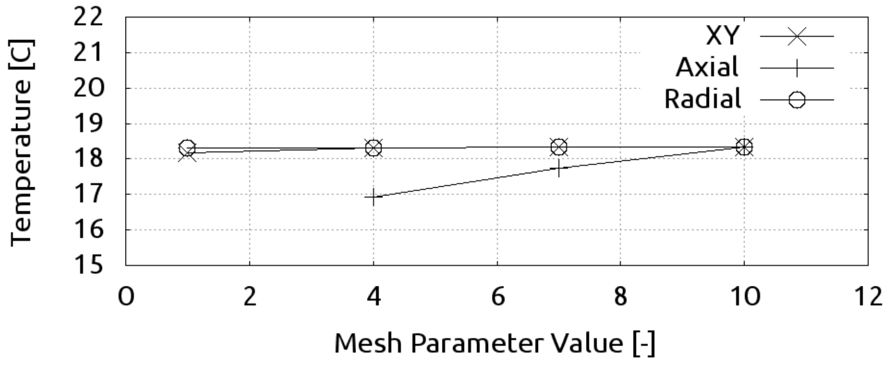
4.3.3.3 Grid Independence

The grid independence study showed that the radial mesh count is relatively insignificant; the XY mesh can provide independence at a low level, whereas the axial mesh count is a major factor. The axial mesh parameter minimum value was therefore set at four. Each of the three output metrics (heat exchanger outlet temperature, basement wall temperature, and basement load) are displayed in Figure 4.9. The data is presented for each metric with three curves. The three curves represent varying a single mesh parameter while the other parameters remain refined at the maximum mesh density value. This isolates the effect to the single parameter being swept.

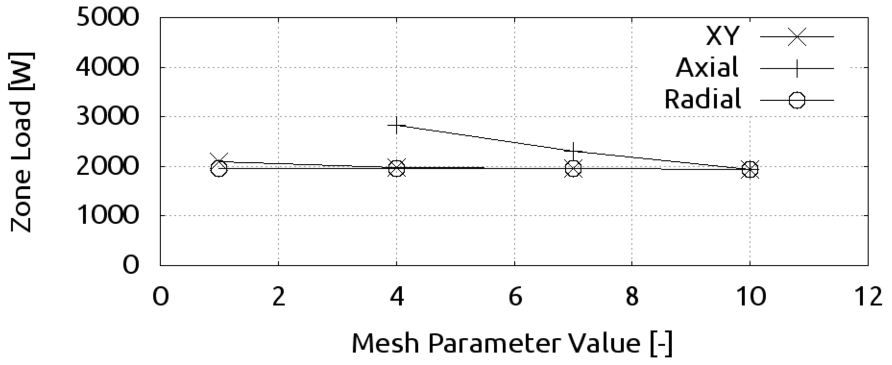
For the heat exchanger outlet temperature (Figure 4.9a), the XY mesh shows a change of nearly 1.7 °C (3 °F) from a single mesh value to the next, but the effects diminish with a coarse grid. The effects of axial and radial mesh parameters provided



(a) Heat exchanger fluid outlet temperature



(b) Basement wall temperature



(c) Basement zone load

Figure 4.9: Grid independence results: value of a domain property as a function of varying each mesh parameter separately

less than a 0.25 °C (0.45 °F) change across the variation of parameters. This confirms the expectation that a single radial cell suffices, while values less than 4 for XY and axial mesh parameters provide independence.

For the basement wall temperature (Figure 4.9b) and basement zone load (Figure 4.9c), the radial and XY mesh parameters are insignificant, showing less than a 10% change throughout the parameter variation. The axial effect is more pronounced, showing variation yet trending toward convergence as the number of axial cells is increased. Since the axial effect did not have an effect on fluid temperature, this indicates that the axial parameter has more effect on the near-zone region, allowing ground temperature variation to be included.

4.3.3.4 Discussion

The results of this study were used to guide the selection of model grid parameters for experimental validation. The model showed greater sensitivity to the axial mesh count than the other mesh parameters. The selected values for XY and radial mesh count was 3, while the axial mesh count was more increased to 12. This value results in a grid where each cell is 3.07 m (10.07 ft) long in the axial direction. Using the coarse grid for XY and radial mesh parameters, while using a refined axial grid results in the computation time displayed on Figure 4.8. This very low computation time is achieved while still producing a high level of accuracy, as demonstrated by further validation in the following sections. Larger values of each parameter could have been selected for experimental validation, however this would result in an increase in computation time and put the model in conditions that may not be feasible for the simulation of systems in practice.

4.3.4 Analytic Validation of Interface Cell

The approach used to model the near-pipe region utilizes a coordinate system interface cell to provide thermal interaction between the two coordinate systems. The energy balance approach used in developing the system of equations to solve the system ensures that under the given assumptions, energy will be conserved. However, the effects of certain assumptions used in developing the interface cell must be validated to ensure the coordinate system mapping can produce suitable accuracy. These assumptions include:

1. The interface cell is spatially isothermal and represented in equation development by any midpoint on the straight sides of the interface cell.
2. The heat transfer between the interface and the inner radial system is one dimensional and driven by the distance between the outermost radial centroid and the midpoint of the side of the interface cell.
3. The heat transfer between the interface and outer cells is Cartesian and driven by the distance between the Cartesian cell centroid and the midpoint of the side of the interface cell.

In order to validate this approach, the pipe was approximated as a line source in an isotropic domain. The idealized simulation domain was constructed with the following properties:

- A single small pipe, centered in the domain
- Domain size $\gg$ pipe size
- Constant ground surface and far-field boundaries ($T = 0$ °C (32 °F))
- Disabled dynamic properties (constant specific heat)
- Initialization of domain at ($T = 0$ °C (32 °F))
- Pipe cell bypassed any fluid flow, a constant heat gain was added to the domain at the pipe wall

In this way, the small pipe approximated a line source in an isotropic domain. This idealization modified the domain boundaries (including the fluid boundary) but left the coordinate system interface treatment unmodified. The analytic solution for the idealized situation was described by Ingersoll and Plass (1948):

$$T\left(r,t\right) = T_0 + \frac{Q'}{2\pi k t}\int_X^{\infty} \frac{e^{\beta^2}}{\beta} d\beta \tag{4.15}$$

Where β is simply an integration variable, and the integral domain limit X is a normalized radius:

$$X = \frac{r}{2\sqrt{\alpha_s t}} \tag{4.16}$$

The numerical model and the analytic solution were sampled at two radial points, both of which were outside of the interface, in the Cartesian domain. One point was close to the interface, at a distance of 0.056 m (.183 ft), while the other point was 0.556 m (1.824 ft) away. The results are shown in Figure 4.10. The simulation domain matched well with the analytic solution, with a peak absolute error of 0.09 °C (0.16 °F). This peak error occurred at the cell nearest the pipe at the initial time step, with the error diminishing rapidly in both time and space away from this point. This is attributed predominantly to the differences between the analytic solution assumptions and the actual model; the pipe was not actually a line source, but rather a small cylinder in the domain.

4.3.5 Undisturbed Ground Temperature

Undisturbed ground temperature was measured at the experimental facility at five depths: 0.3 m (1 ft), 0.6 m (2 ft), 0.9 m (3 ft), 1.5 m (5 ft), 01.5 m (6 ft). At the shallowest measurement, the ground temperature is strongly dependent on surface conditions such as solar gain, evapotranspiration, and convection to outdoor air. As the depth increases, the temperature becomes less dependent on surface effects, and more

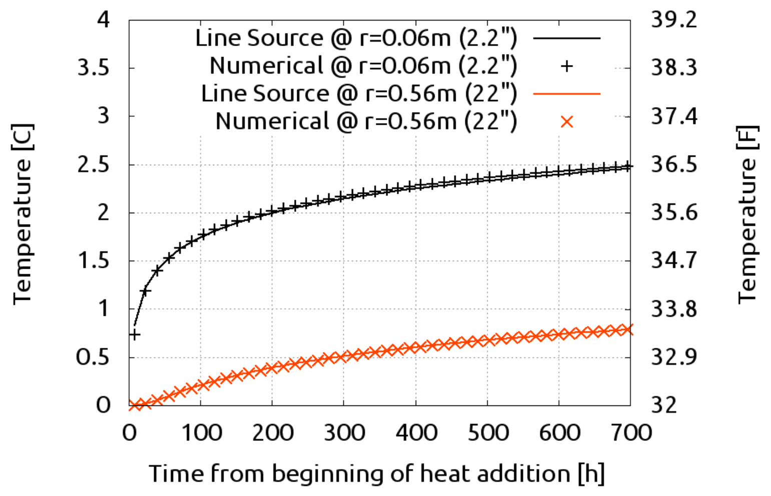
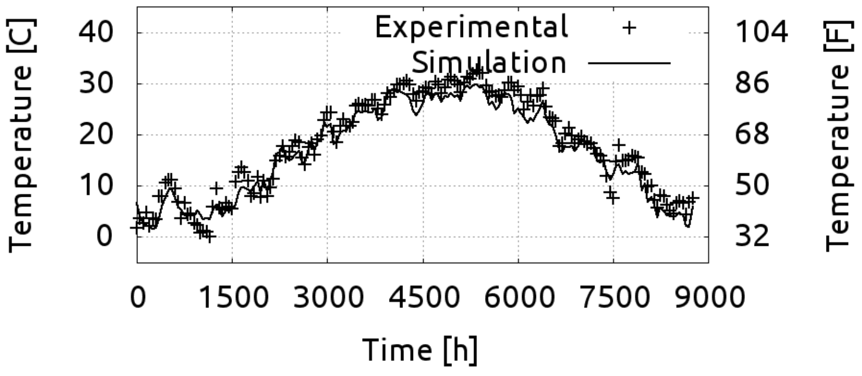


Figure 4.10: Comparison of analytic and numeric temperatures for validating the accuracy of the coordinate system interface cell

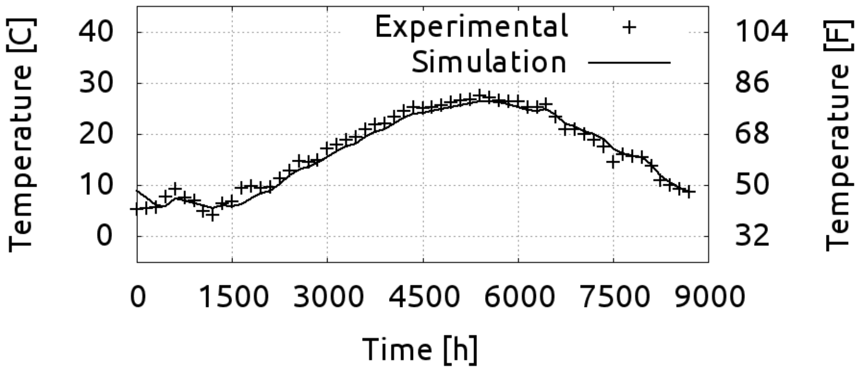
dependent on deep ground effects. In terms of simulation, these include the selection of far-field boundary condition models and parameters.

Parameter estimation was performed using this experimental data to optimize simulation parameters. A cyclic heuristic direct search algorithm was employed where the objective function was the sum of the squared error between experimental and model data. This algorithm is robust if given a valid starting point for the optimization. The decision variables in the study were the ground density and specific heat, and the far-field temperature specification parameters $\left(\overline{T}_s, \overline{\Delta T}_s\right)$. The feasible ranges on the parameters are approximately 20% of the initial starting point. The initial starting point for thermal properties of the soil are based on a clay loam soil with water content as described by Lamberg et al. (2004). The initial starting point for temperature data is approximated from measured weather data. The parameter estimation procedure provided the values shown in Table 4.1.

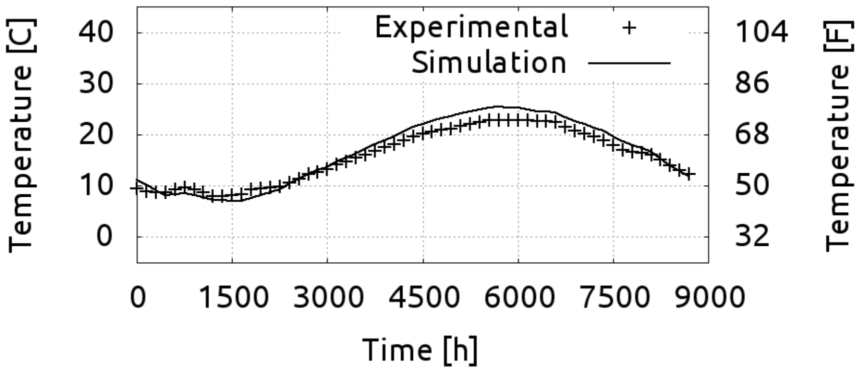
Using these optimized parameters, the undisturbed ground temperature was predicted by the simulation model without any pipes in the domain. The results for three representative depths are shown in Figure 4.11. The mean bias error over the entire data set was 0.36 °C (0.65 °F).



(a) $Depth = 0.30$ m(1 ft)



(b) $Depth = 0.91$ m(3 ft)



(c) $Depth = 1.83$ m(6 ft)

Figure 4.11: Undisturbed ground temperature results at multiple depths below the ground surface using optimized (parameter estimation) parameters.

At greater depths, the model deviated more from the experimental measurements than at the shallower depths. A possible source of error is the far-field boundary temperature formulation (Kusuda and Achenbach, 1965). This form of the boundary condition may not capture all of the boundary effects that may exist in the experimental data, including:

- Unusual variation in seasonal temperature variation in the previous year(s)
- Non-isotropic ground, perhaps layers of different ground materials
- Proximity to underground water table and ground water flow
- Other experimental artifacts (ground not actually *undisturbed*)

4.3.6 Component-level Validation

Component model validation was completed to demonstrate the model's ability to predict outlet conditions provided a tightly bounded solution domain. The entering fluid temperature was fixed at each time step to experimentally measured heat exchanger inlet temperature, which ensured that over the course of the simulation, the error in total heat transfer to the ground was minimized. By controlling the amount of heat transfer into the ground, the boundary condition for the fluid remained accurate and did not drift from the experimental conditions.

The simulated heat exchanger outlet temperature matched experimental data with a mean error of 0.3 °C (0.54 °F). The quality of the component-model validation is better represented with the magnitude of the temperature change across the heat exchanger, or heat transfer rate. Assuming a constant specific heat, this was calculated

Table 4.1: Simulation parameters from optimization against experimental undisturbed ground temperature

Parameter Name	Symbol	Value	Units	Value	Units
Ground Density	ρ_s	852.3	kg/m^3	53.207	lb/ft^3
Ground Specific Heat	$C_{p,s}$	2073.8	J/kgK	0.49	$btu/lb°F$
Average Annual Surface Temperature	$\overline{T}_s$	12.86	°C	55.16	$°F$
Avg. Ampl. of Surf. Temperature Variation	$\overline{\Delta T}_s$	13.73	°C	24.71	$°F$

as:

$$\dot{q} = \dot{m}C_p \left(T_{out} - T_{in}\right) \tag{4.17}$$

The resulting heat transfer rate is shown in Figure 4.12. With a tightly controlled (fixed inlet) simulation, the predicted heat transfer rate matched the experimental data with a mean bias error of 27.5 W (93.8 btu/h).

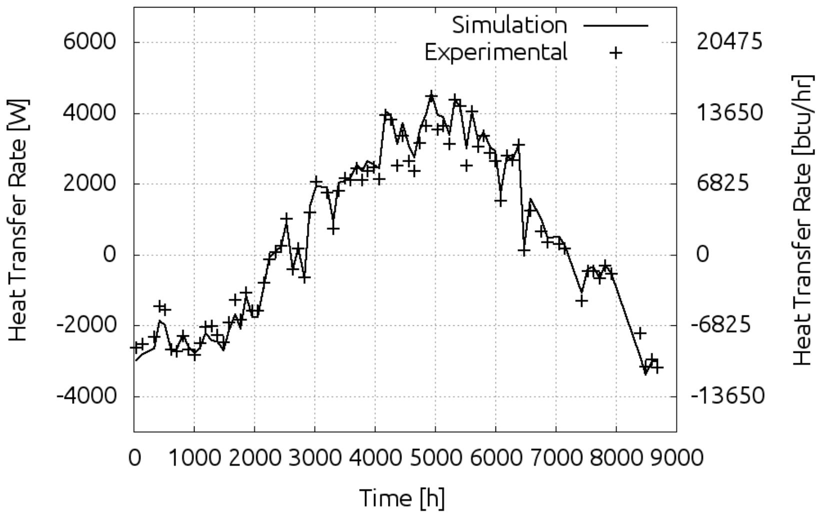


Figure 4.12: Daily averaged heat heat exchanger heat transfer rate validation using experimental measured heat exchanger inlet temperature

As shown in Figure 4.12, the model predicted individual peaks of heat transfer rate with good accuracy aside from deviations in the initial and peak heat rejection periods. The deviation in the initial period is possibly due to the initialization of the ground domain, which may be significantly different from that found at the experimental site in the back-filled soil. The undisturbed ground temperature prediction also could not match experimental measurements in the peak heat rejection period. The error in heat transfer rate prediction in this region is expected to be due to this effect, which may be manifested as an error in thermal properties or boundary parameters.

4.3.7 System-level Validation

For system simulation, the load on the heat exchangers was calculated from experimental data, and used as a boundary for a full loop simulation. This type of validation tends toward lower accuracy than the component-level validation because the boundary conditions on the fluid thermal network are not at the inlet of the heat exchanger model, rather they exist as boundary conditions on the fluid loop. Any inaccuracy in heat transfer from the fluid to the ground affects the fluid response in subsequent time steps.

The heat pump entering fluid temperature (same as heat exchanger outlet temperature within the simulation model) is shown in Figure 4.13. The mean bias error in outlet temperature prediction was 1.3 °C (2.3 °F). The model showed less accuracy predicting temperatures beginning near hour 6500 when the system was off-line periodically. When there is no flow in the system, the fluid temperature is predicted using a simplified natural convection approach. As shown in Figure 4.13, the model tends to under predict these periods. Once flow is restarted, the fluid heat transfer is governed by the loads in the system and the forced convection model.

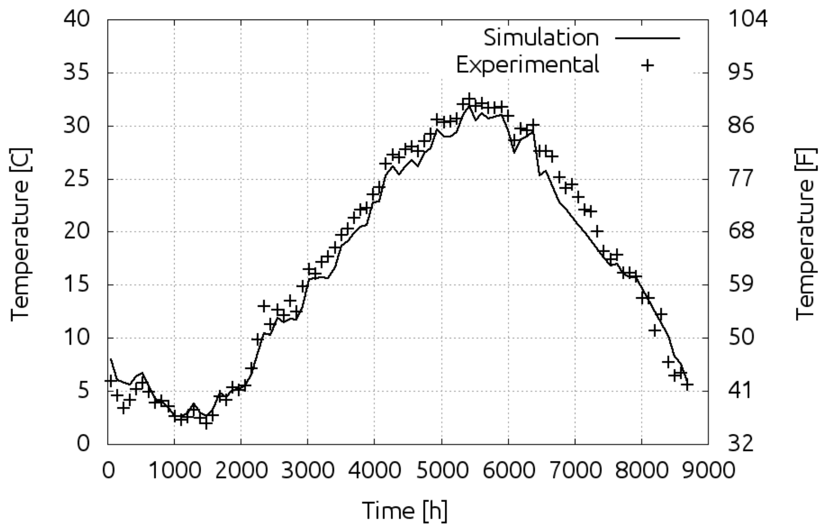


Figure 4.13: Daily averaged heat pump entering fluid temperature validation using experimental heat transfer to drive a full system simulation

4.3.8 Basement Wall Heat Flux

The experimentally measured data at the foundation heat exchanger test site includes basement wall heat flux data. For foundation heat exchangers specifically, the thermal exchange with the zone is an important design parameter. The proximity of the heat exchanger pipes has a significant impact on the zone loads and the zone conditions. In EnergyPlus, building surfaces (walls) are defined as single objects, modeled with one-dimensional transient conduction. Accordingly, for this validation the basement floor and wall were single surfaces.

Wall heat flux was measured experimentally at three locations along the basement wall [depths = 0.36 m (1.17 ft), 1.07 m (3.5 ft) and 1.73 m (5.67 ft)]. An area-weighted averaging scheme was used to regress these experimental values into a single representative wall heat flux measure. Measurement zones were established for each measured point. The centroid between the measurements was used as the interface from one measurement zone to the next. The measured value represented the heat flux for the entire zone. The area fraction of each zone was used to define the weight of each measurement when averaging them together. This is described visually in Figure 4.14. The top averaging interface sits centered between measurements 1 and 2, while the bottom averaging interface sits between measurements 2 and 3. This resulted in normalized weighting factors for each measurement value: measurement #1 has a weight of 0.28, measurement #2 has 0.27, and measurement #3 has 0.45.

The averaged wall heat flux was then calculated:

$$\overline{\dot{q}''} = 0.28\dot{q}_1'' + 0.27\dot{q}_2'' + 0.45\dot{q}_3'' \tag{4.18}$$

With this averaging performed, the resulting average measured wall heat flux was compared to the simulation wall heat flux for the entire surface, as shown in Figure 4.15. The overall trend and peak heat transfer was predicted by the model

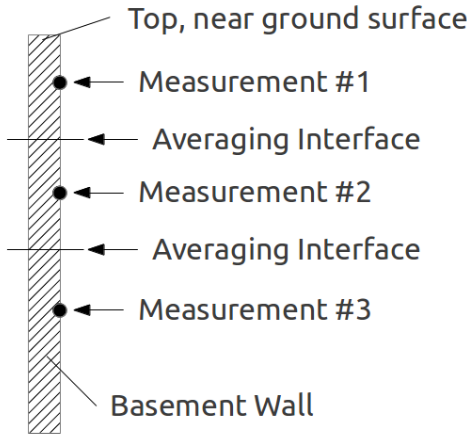


Figure 4.14: Description of averaging approach for multiple experimental measurement locations of basement wall heat flux

with an average annual absolute error between the model and experimental data of 1.1 W/m^2 (0.35 btu/hft^2). The model shows higher fluctuations, which represents a higher sensitivity to the ground surface phenomena than the experimental top measurement value.

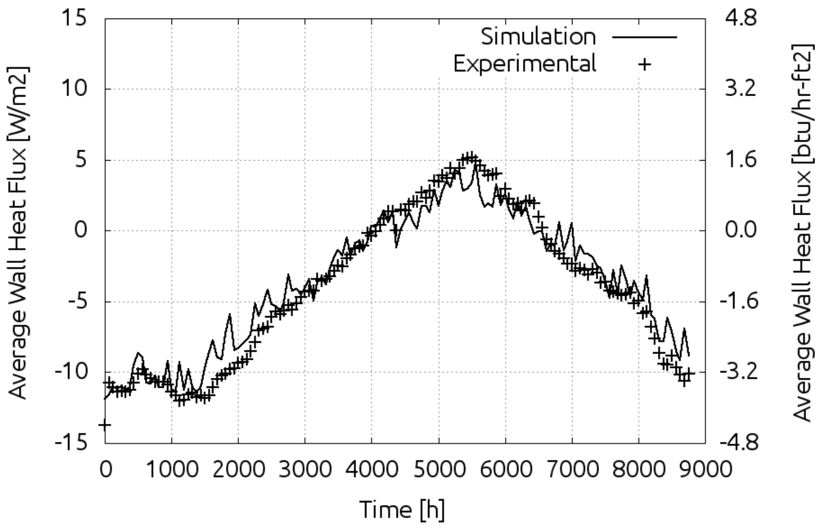


Figure 4.15: Validation of basement wall heat flux against experimental data

4.4 Special Development

Additional features were implemented or studied as a part of the final stage of this research project. These include the following:

- Implementation of a U-Tube borehole model
- Implementation of an improved fluid natural convection model
- Further grid sensitivity analysis

4.4.1 U-tube Borehole Model

The initial piping system model implemented in the program was designed to handle a single pipe within any Cartesian cell. This alone is sufficient for handling a wide variety of heat exchanger applications, including single tube run-around systems, foundation heat exchangers, and any other where the inter-pipe spacing is larger than the scale of single pipe diameters. In these cases, Cartesian mesh can be placed between pipes to capture the heat transfer between them. However, in cases where the pipes are placed much closer, and perhaps embedded within a different material such as grout, a specialized model has been implemented. The assumptions used in this specialized model include:

1. Exactly two pipes are placed within a uniform material (grout).
2. The pipes are spaced a specific distance apart which is constant for the entire length of the borehole, and characterized by a shank spacing.
3. The flow in the pipes is in opposite directions, implying a u-bend at one end of the domain.
4. The u-bend at the borehole is idealized to the point of being adiabatic, as fluid is passed from the outlet of one leg directly to the inlet of the other leg.

This u-tube borehole model fits into the main horizontally-oriented Cartesian domain in a fashion similar to the single pipe model. Because of this, it is most suitable for horizontal borehole applications. While these applications are not very popular in practice, this arrangement is useful because high quality experimental validation data is available for such a configuration. Future model development includes adjusting the boundary condition specification to allow vertical boreholes to be simulated, which is a much more common application. The expansion for vertical boreholes is unrelated to the borehole itself, so a valid horizontal borehole model is expected to be a suitable vertical borehole model as well.

4.4.1.1 Introduction

In order to fit with the rest of the model formulation, the borehole heat transfer model must, at a minimum, be able to track fluid temperature in *each* pipe from inlet to outlet. The overall structure of the borehole as it sits within the outer simulation domain is shown in Figure 4.16. The borehole model exists within the radial system already established for single pipes, but uses a specialized thermal network. Thus, the interaction with the outer Cartesian system is no different than the regular single pipes. Key aspects of the model include:

- Tracking fluid conditions through both legs of the borehole
- Properly interacting thermally with the inner radial soil mesh
- Capturing variation in properties when considering the grout region

To characterize the responsibilities of a single borehole cell within this axial discretization, a single cell is extracted, and the cross section is shown in Figure 4.17. Note this shows some of the boundary conditions which are imposed on the governing equations in the system. Most notable for the current discussion is the addition of a second entering fluid boundary condition. Aside from this, the addition of grout and multiple pipe segments is simply an addition to the set of transient conduction equations already being solved on each cell.

In order to simulate the borehole internal phenomena, a thermal network is established and solved numerically, as with the rest of the simulation domain. Model formulations have been proposed previously. Xu (2007) utilized a lumped technique where both the inlet and outlet legs of the system are lumped into a single tube with equivalent thermal properties. This works well in that study, as the simulation

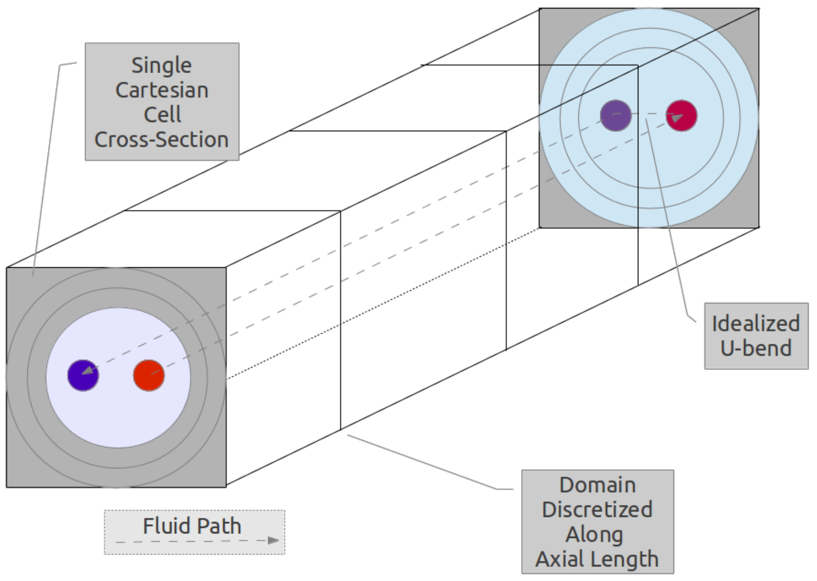

Single Cartesian Cell Cross-Section
Idealized U-bend
Domain Discretized Along Axial Length
Fluid Path

Figure 4.16: Axial borehole visualization, showing fluid path through u-bend, axial discretization, and numerical grid within axial cross section

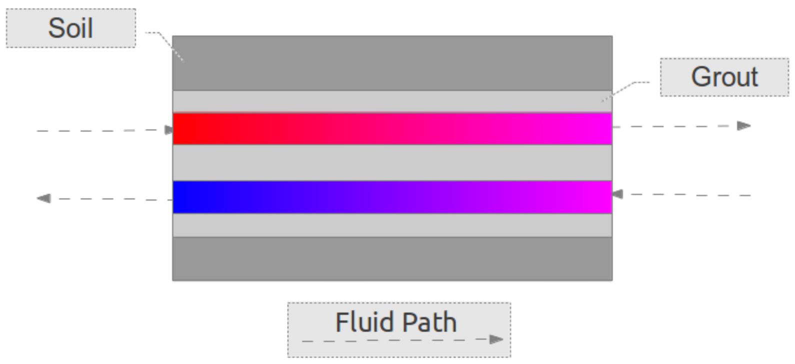


Figure 4.17: Single borehole cell longitudinal cross section

domain does not rely on tracking fluid temperatures in an axial mesh. For the current simulation model this would not useful; it would require an unnecessary level of bookkeeping to back out individual fluid temperatures. Claesson and Hellstrom (2011) describe a multipole method to solve for the steady state heat transfer phenomena in boreholes with generalized pipe placement and sizing. The initial assumption of steady state heat transfer phenomena makes it unattractive considering the otherwise transient nature of the simulation model. However, this method is left as an alternative approach for future work to allow more precise resistance networks to be established.

4.4.1.2 Model Development

The borehole is modeled using a transient governing heat conduction equation. The equation is solved at each thermally distinct entity in each cross section of the borehole and eventually the fluid system is also solved flow-wise from the inlet to the u-tube through the idealized u-bend and finally to the outlet of the u-tube. The borehole model sits within a radial system of soil nodes, which themselves exist within an interface to the outer Cartesian coordinate system. The interaction with the outer Cartesian system can be found in section 4.2.2 and is not re-discussed in this section.

The cross-sectional mesh for the borehole model is shown in Figure 4.18. Note that at a single cross section, there are two cross sections for each of fluid and pipe material, as well as a special grout interface. In a single pipe section, while the domain is two-dimensional, the heat transfer within the radial section is modeled as one-dimensional. This assumption was present for cells with only a single pipe, as the pipe was centered radially within the rest of the radial system. It is applied here with an understanding that the resistance calculations may have some uncertainty in relation to this.

As described in the introduction, models have been developed previously, but

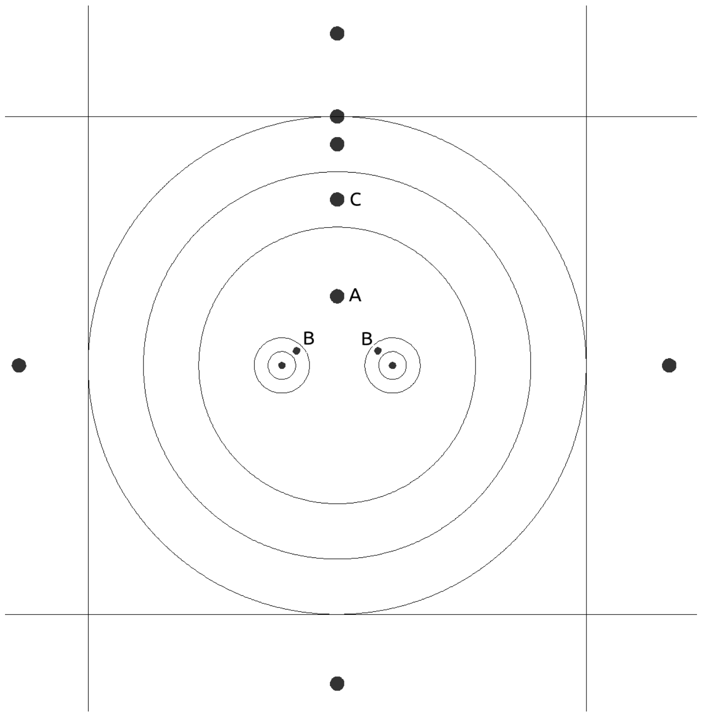


Figure 4.18: Ground heat exchanger borehole model grid

have either significant limitations or are too detailed to be useful in this project. The context of this entire modeling work has been providing flexibility and robustness while minimizing burden and detail. To continue this, the thermal network between the grout, innermost radial soil cell, and pipes is modeled as linear one-dimensional Cartesian based solely on the distance between the nodes, as shown in Figure 4.18. In this Figure, points labeled "B" represent the radial centroid of each pipe material. Point A represents the *defined* location for the grout cell node. In the orientation shown in Figure 4.18, this point is centered horizontally in the borehole cross section, and oriented vertically halfway between the radial center and the top borehole wall. Although the last sentence used the terms horizontal and vertical, the one-dimensional assumption used in the thermal network development blurs the model's representation of the physical system. This approach can also be presented as an adjusted set of thermal properties, which is a common approach to modeling thermal networks of complex geometry.

The simulation of the borehole cell itself is performed in a two-step procedure.

The first step consists of simulating the radial soil and grout system. This is followed by a flow-wise simulation of the pipes and fluids through the inlet and outlet legs of the borehole. The actual operation is characterized in Algorithm 1.

Algorithm 1: Overall flow of solution algorithm for a single u-bend borehole

```
1  for $c \in Segment.Cells$ do
2      Update Radial Soil and Grout
3  end
4  Prepare Borehole Inlet Temperature
5  for $c \in Segment.InletLeg.Cells$ do
6      Update Inlet Leg Fluid and Pipe
7      Prepare Temperature for Next Cell
8  end
9  Pass Temperature From Inlet Leg to Outlet Leg
10 for $c \in Segment.OutletLeg.Cells$ do
11     Update Outlet Leg Fluid and Pipe
12     Prepare Temperature for Next Cell
13 end
```

Algorithm 1 is a simplified overview. The radial soil and grout simulation call on line 2 is summarized in Algorithm 2. Also from Algorithm 1, the operations in lines 6 and 11 are similar, and summarized in Algorithm 3.

Algorithm 2: Iteration loop for updating the grout and soil cells which interface the borehole grout to the outer Cartesian system

```
1  while Not Converged do
2      Shift Relevant Temperatures For New Iteration
3      Simulate Radial-Cartesian Interface
4      Simulate Radial Soil Cells
5      Update Grout Temperature
6  end
```

The final model formulation utilized a significant assumption related to grout thermal properties. It was found during experimental validation (next section, 4.4.1.3) that using standard thermal properties with the thermal network set up here that the effects of the grout thermal storage were significantly over-accounted. The grout thermal mass was causing an unrealistic amount of lag in the fluid temperature dis-

Algorithm 3: Iteration loop for updating the pipe and fluid temperatures for a single u-bend borehole in a flow-wise fashion

```
1 while Not Converged do
2     Shift Relevant Temperatures For New Iteration
3     Simulate Pipe Cell
4     Update Fluid Temperature
5 end
```

tribution prediction. This is explained by a qualitative description of the thermal phenomena occurring in the borehole. The thermal network assumes that the heat leaving the pipes must traverse fully through the center of the borehole before propagating out of the borehole wall. This causes a high amount of heat being stored in the grout. However, the heat transfer is more complex than this, and a significant portion of it will pass directly from the pipe wall to the borehole wall, bypassing a majority of the grout. It was found that this thermal network model works well with experimental data when the grout thermal storage is eliminated (by reducing multiple orders of magnitude compared to other nearby features). The effects of this assumption will be dependent on the grout thermal conductivity and pipe spacing, but was able to match experimental data with an extremely high level of accuracy as described in the following section. This conclusion is in accordance with assumptions commonly used according to Claesson and Hellstrom (2011), as well as the a common approach used in cases like these that utilize adjusted thermal properties to account for various geometric attributes and multidimensional heat transfer.

4.4.1.3 Experimental Validation

The borehole model was validated using experimental data from a well-documented borehole test (Beier et al., 2011). This document includes a detailed overview of the experimental setup, and the article includes attachments that contain the experimentally measured data in spreadsheet form. This is an ideal source for performing experimental validation of this model.

The experimental setup consisted of a rectangular sandbox containing a u-tube borehole. Boreholes are most commonly installed in a vertical fashion, although this experimental setup has the sandbox and borehole oriented horizontally. In this way, the entire borehole can be contained inside a building to allow tight control of the boundary conditions on the sandbox exterior, and provide a tightly-controlled experimental data set.

Nearly all experimental parameters required for model validation were directly provided by the article, leaving only a few to be estimated. A summary of the input parameters used is provided as Table 4.2. The parameters which were not explicitly available include the dynamic thermal properties (ρ, C_p) of the sand and grout. The pipe dynamic thermal properties were not prescribed, but these values do not change much under small temperature changes. The boundary and initialization temperature can be estimated from plots in the paper, so this was done as an initial validation, though an optimization was performed to optimize these temperatures.

In addition to these static values, the system heat addition and flow rate were observed to be slightly dynamic during the initial experimental time. The data was averaged hourly and the results are shown in Figure 4.19. Although the variation is minimal in the long run, and overall average values may have provided suitable results, using these transient values ensured that the simulation conditions matched experimental conditions as much as possible, especially in the early periods where the temperature is changing most rapidly.

Figure 4.20 shows the results of validation against a thermal response test using normal axes. There are two result series shown on the plot. These results were obtained after the approximation described previously was employed, in which the thermal storage of the grout was eliminated from the simulation because of poor model performance (grout resistance effects are still present).

The initial results (estimated temps) in Figure 4.20 were based on using estimated

Table 4.2: Summary of parameters used in validating the borehole model, where "Sandbox Paper" represents Beier et al. (2011)

Variable	Value	Units	Source	Comments
Domain Size X	1.8	m	Sandbox Paper	Overall sandbox size
Domain Size Y	1.8	m	Sandbox Paper	Overall sandbox size
Domain Size Z	18	m	Sandbox Paper	Overall sandbox size
Sand k	2.82	W/mK	Sandbox Paper	Provided by paper
Sand rho	1400	kg/m^3	Nominal Property	Nominal volumetric split between density and specific heat
Sand Cp	1400	J/kgK	Nominal Property	Nominal volumetric split between density and specific heat
Boundary Temp	[varies]	°C		Initially estimated from paper data, then optimized
Initialized Temp	[varies]	°C		Initially estimated from paper data, then optimized
Pipe k	0.39	W/mK	Sandbox Paper	Provided by paper
Pipe rho	950	kg/m^3	Nominal Property	HDPE Pipe
Pipe Cp	1950	J/kgK	Nominal Property	HDPE Pipe
Grout k	0.73	W/mK	Sandbox Paper	Provided by paper
Grout rho		kg/m^3		Grout mass is adjusted to accommodate model assumptions
Grout Cp		J/kgK		Grout mass is adjusted to accommodate model assumptions
Pipe ID	0.02733	m	Sandbox Paper	Provided by paper
Pipe OD	0.0334	m	Sandbox Paper	Provided by paper
Borehole OD	0.126	m	Sandbox Paper	Aluminum pipe represents borehole wall in sandbox study
Pipe Center	(0.9, 0.9)	m	Sandbox Paper	Centered in sandbox
Shank Spacing	0.0196	m	Sandbox Paper	Calculated from center-to-center spacing and pipe size

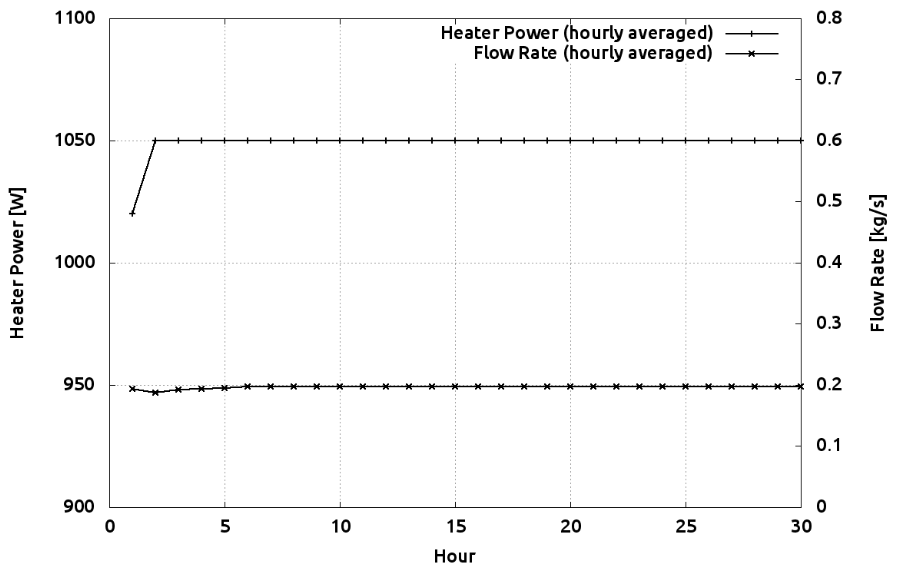


Figure 4.19: Ground heat exchanger borehole model dynamic validation input parameters

boundary and initial domain temperatures, as there is some uncertainty as to the exact values utilized in the paper's experimental work. Even with this estimation, the results show a high degree of accuracy given the model's simplicity and assumptions in the thermal network. However, to determine if the model could perform better with further accuracy in input parameters, an optimization was employed to vary the domain initial and boundary temperatures and achieve a better match to the thermal response test data. The results after the optimization were indeed better than the estimated data, however either case is suitable given the uncertainty in other parameters.

Figure 4.21 shows the same results, but using a log axis, to coincide with the presentation of results in the paper. The results are the same in that the estimated results are a very good match, and the optimized results push the results even closer.

4.4.1.4 Summary

The borehole model utilizes many simplifications which are in place due to the ultimate goals of broad usability and efficiency in input parameters and computation.

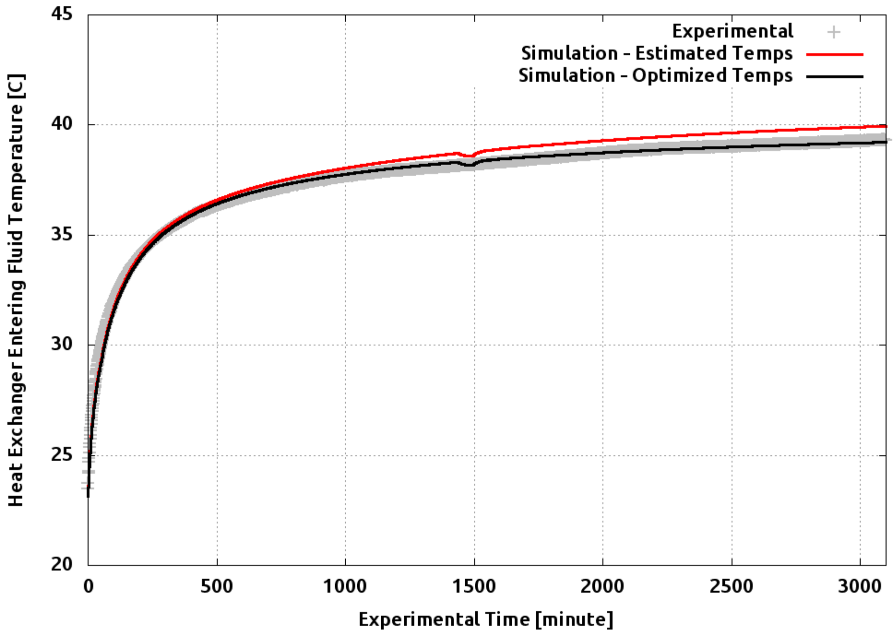


Figure 4.20: Validation of borehole model using sandbox thermal response test data, normal plot

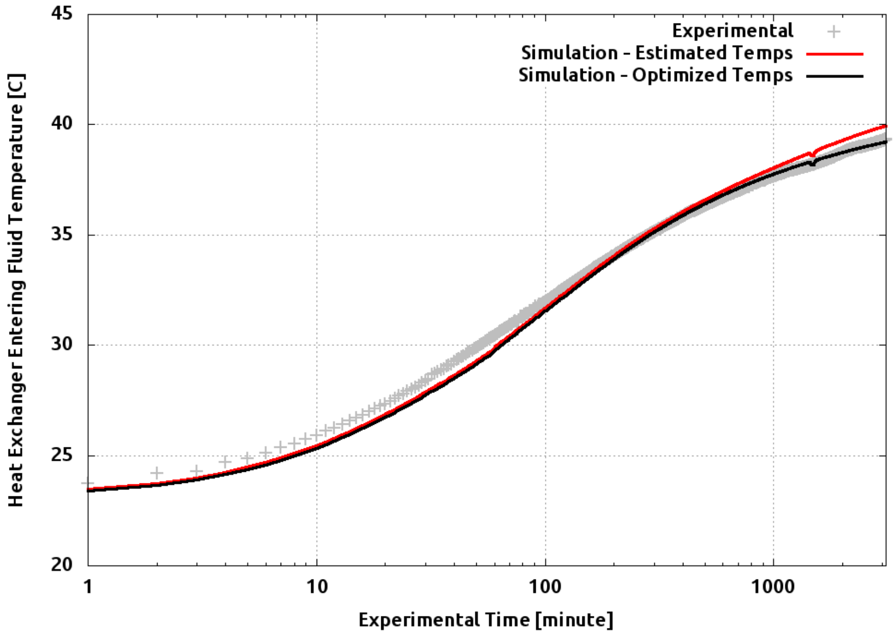


Figure 4.21: Validation of borehole model using sandbox thermal response test data, log plot

Even with these simplifications, the model was able to produce a high degree of accuracy through experimental validation of a thermal response test using a horizontal borehole. During model development, the work by Claesson and Hellstrom (2011) was described, but not utilized because the assumptions in the model seemed to be too constraining given the transient nature of the rest of the model. After experimental validation was achieved with a simplified thermal network that makes the grout essentially steady state, it would now be interesting to test whether the multiple resistance network would be as suitable a candidate as the currently implemented version. Further work on this remains as possible future work.

4.4.2 Improved Fluid Natural Convection Model

During experimental validation of the ground heat exchanger model using foundation heat exchanger data, the model performed well but showed some deviation during times when the system was not running, and immediately after the system started back up. It was thought that the effects may be due to the natural convection correlation utilized during system "off-cycles." To test this, the natural convection coefficient was varied significantly during test runs and the model was found to be insensitive to this parameter. As such this is not being studied further. The deviation may simply be due to the assumptions built in to the model, including a lack of moisture transport.

4.4.3 Enhanced Grid Sensitivity Study

A grid independence study was performed previously as described in section 4.3.3.3. During this study some preliminary results were found regarding the importance of certain grid parameters.

Further grid study was performed in an effort to clarify the sensitivity of the model under certain grid conditions. Two model configurations were tested in this process:

1. A two-pipe ground heat exchanger model, where two pipes are positioned beside each other horizontally, with only the ground domain surrounding
2. A full foundation heat exchanger simulation model, with 6 pipes and interaction with the basement zone

The same mesh parameters utilized in the previous study were reused in this study:

- XY Mesh Density: A measure of the mesh density in the X and Y directions (longitudinal to the pipe cross section)
- Axial Mesh Density: A measure of the axial discretization of the domain
- Radial Mesh Density: The number of cells to be utilized within the radial mesh, between the pipe and the outer Cartesian domain

In the two-pipe ground model, the annual average heat pump entering fluid temperature is used as a reference model output value. For the two-pipe model, there was a distinct lack of sensitivity to the axial discretization, as shown in Figure 4.22. In this figure, each horizontal set of data points represents a single combination of radial/XY mesh values, which is then swept over each axial value. For very low grid counts in the axial direction, there is some variation for each row, however after about an axial count of 10 the results do not change significantly. The entire y-scale of the plot is only 1.4 °C, so even at the initial variation, the change is much less than 0.1 °C.

Showing independence in the axial direction is important to ensure applicability of the fluid cell governing equation in this discretized domain. This implies that the axial grid count can be kept to a reasonable value without requiring enormously large axial cell counts to achieve suitable grid independence.

The next test is for the XY mesh density parameter. The results are presented in Figure 4.23 in a similar fashion to the axial mesh.

The XY results in Figure 4.23 reveal that for any combination of axial/radial grid parameters, the grid becomes independent of the XY mesh density with a value

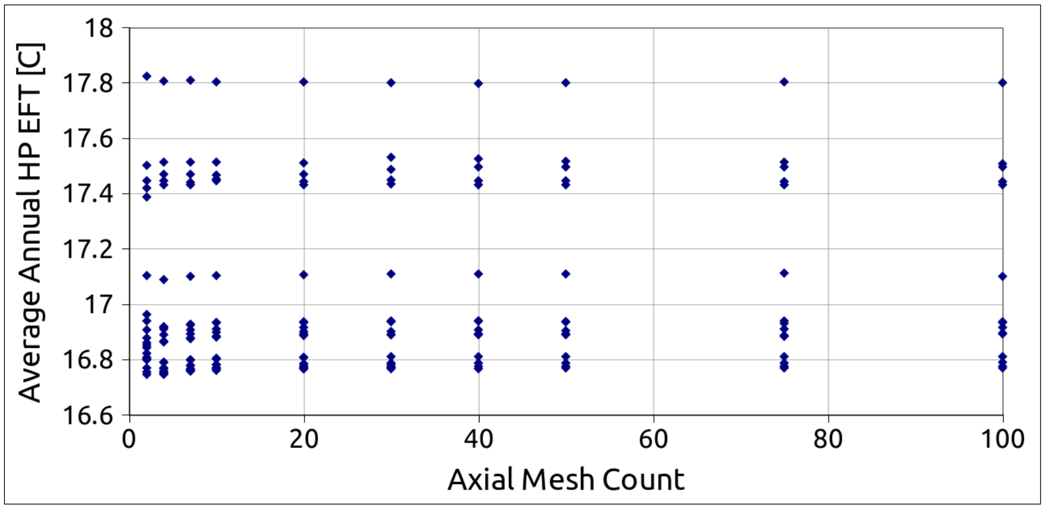


Figure 4.22: Further grid sensitivity study, “two pipe model” results as a function of axial mesh count

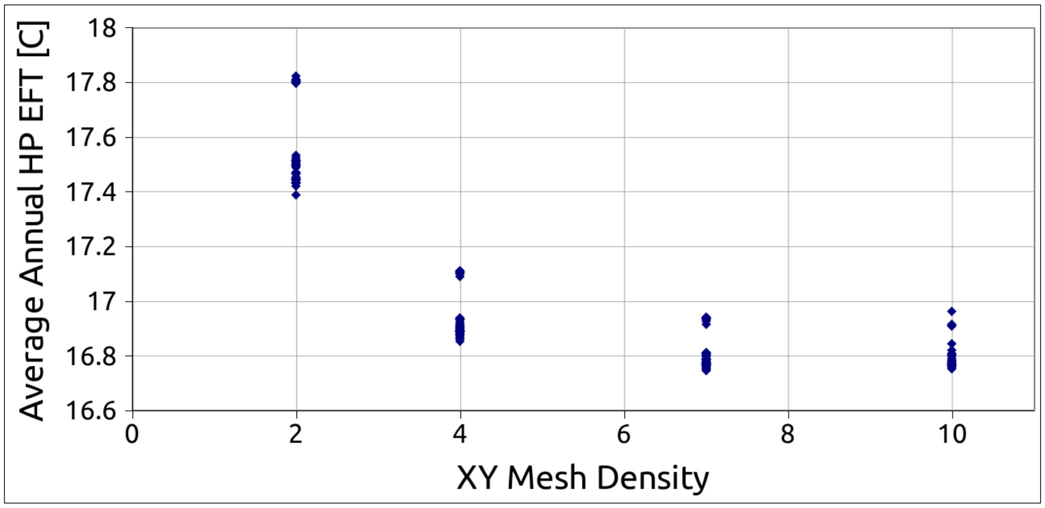


Figure 4.23: Further grid sensitivity study, “two pipe model” results as a function of XY mesh density

of 7, as the results at 10 are very close to the results at 7. For a two-pipe model such as this, the resulting Cartesian mesh at a given axial cross section will contain around 315 cells, which makes for a reasonable amount of computational burden for simulating these systems.

The final mesh parameter to be swept in this study is the radial mesh count. This value defines the number of radial cells to be utilized as an interface to the Cartesian domain. With a higher radial mesh count, the grid density will be much higher around the pipe, and theoretically able to capture higher order gradients than with fewer cells, which would tend to blur the gradients. In this test, the model was only slightly sensitive to the radial mesh density around the pipe up until a value of about 20. Beyond 20, the cells became extremely small, and the results began to drift away from a converged point. This implies that a numerical artifact, likely related to truncation error compounding in such cases.

In the foundation heat exchanger case, even with a small mesh count in the X-Y direction, the domain contained a high number of cells, just due to the mesh partitions (pipes). Because of this, the XY parameter was an insignificant parameter. The radial mesh was also found to be insignificant similar to the two-pipe configuration with reasonable number of radial cells. However, this test revealed an instability present in the zone interaction. For axial mesh counts less than around 30, the model produces suitable results and is not highly sensitive to the parameter. However, above this amount, the model begins to show instability with unrealistic cell temperatures and interaction with the basement zone. This is likely a combination of two coupled effects:

- The coupling between the ground domain and the zone is managed by a specialized surface outside boundary condition, called an other-side-conditions model in EnergyPlus. This allows a boundary temperature and convection coefficient to be specified for any surface. For this model, since the ground domain is assumed to be in direct contact with the surface, the convection coefficient is set to a very high value in order to approximate a fixed surface temperature. In most simulation cases this appears to be suitable, however the artificial convection coefficient appears to break down as the cell count becomes larger, cell sizes shrink, and the relative error increases.

- This effect may be more pronounced in cases with large cell counts because of computational rounding error within the program. Using an artificially large value of convection coefficient may be causing the results to become truncated, and this is compounded as the number of cell calculations increases.

The solution to this problem involves a closer look at the coupling between the two different model domains, and an improved communication mechanism at this interface. This is left as future work for this model. In the end, this grid sensitivity study resulted in a single conclusion:

Each mesh configuration requires an independent grid independence study.

While the resulting mesh has been coarse and still provided satisfactory results under experimental validation studies, the dependence on individual mesh parameters may vary between configurations.

4.5 Conclusions

A generalized horizontal ground heat exchanger model has been developed which integrates systems within a whole building energy simulation environment. The model uses a coarse grid three-dimensional Cartesian coordinate system as the basis for a numerical solution, with the near pipe regions meshed using a secondary radial coordinate system. This approach provides a refined grid in the near pipe region, and is generalized to allow any number of pipes to be placed in the domain. Fluid flow in the pipe is simulated in a flow-wise fashion as it circuits through the domain to capture interference effects of multiple pipes and flow direction.

The model is integrated with the zone heat balance through a boundary condition at the zone exterior surface. The model is also coupled to the hydronic system simulation through the fluid inlet and outlet of each fluid circuit in the model. These integrations allow the same mass of ground to interact thermally with the zone and the ground heat exchanger that may be serving the zone. This allows for studies of near-zone heat exchangers with improved accuracy over decoupled approaches. The model provides suitable accuracy with a coarse grid when validating against experimental measurements. Heat exchanger exiting fluid temperature is predicted with a mean bias error of 1.3 °C (2.3 °F). Average annual basement wall heat flux is predicted to 1.1 W/m^2 (0.35 btu/hft^2).

In the last stage of this research work, further work was performed including the development of an experimentally validated, simplified two-pipe borehole model, and investigations of fluid natural convection effects and grid sensitivity.

A development snapshot of the source code for this simulation model is provided in the appendix of this document. This code is for a standalone version of the model, not the version implemented in EnergyPlus. The EnergyPlus source code has, at the time of this writing, been released under an open source license, so the full EnergyPlus-coupled code is available from appropriate sources. However this standalone code may be used as a simpler test and development environment as it contains much less complexity than the EnergyPlus version.

CHAPTER 5

Summary

5.1 General Summary

The ultimate goal of this work has been to develop a new ground heat exchanger model that is capable of producing sufficient accuracy in applications where inter-domain effects are important. These effects include the heat transfer interaction between a zone heat balance and the ground, and also the interaction between the ground and a full central plant simulation. The ground heat exchanger model was developed and validated against a number of conditions, producing quality results even with a very coarse grid.

The ground heat exchanger model was implemented as a component model for a central plant simulation engine inside EnergyPlus. To ensure the central plant was sufficiently robust and accurate to simulate the ground heat exchanger in a number of applications, the central plant required a new solution algorithm. A new loop solution algorithm was proposed and implemented inside EnergyPlus which improves the reliability and flexibility in simulating not only ground heat exchanger configurations, but also chilled water loops, hot water loops, condenser loops, and any number of other diverse configurations.

To ensure that the entire system was suitable for as many applications as possible, an investigation into the effects of transport delay was performed. Experimental data was measured to support a modeling study that showed that different modeling techniques can produce significantly different system responses. Bounding studies implemented in a full whole building energy simulation implied that whole building energy effects are not sensitive to a particular transport model, though the effect of transport is expected to be significant in loops with more detailed temperature-dependent component models, than in idealized simulation loops.

5.2 Conclusions

A number of conclusions have been reached during this work:

1. A flexible, robust central plant simulation algorithm, suitable for many in-practice and novel component configurations, can be obtained without requiring a the computational burden and high level of input specification of a detailed pressure network solution.
2. There is an abstraction process required to properly model hydronic systems with this solution algorithm. This process requires analysis of the intention of the loop components and controls; it is not obvious from the loop topology alone.
3. The effect of transport delay on whole building energy use is less dependent on the transport model itself and more dependent on the remaining loop configuration. Downstream components on the loop tend to blur the effects of a delay component. This effect is amplified on loops that consist of predominantly idealized components. The more sensitive a loop is to the temperature variation, the more sensitive it will be to the effects of a transport delay object.
4. The ground heat transfer phenomena in a ground heat exchanger model can be predicted with a suitable level of accuracy using a generally coarse major grid, but with a refined grid in the regions of highest activity. For this model, that includes both a refined major grid structure, as well as a secondary coordinate system established in regions near heat exchanger pipes.
5. Surface heat balance effects are a critical part of accurately modeling shallow ground heat exchangers, with a significant sensitivity on the evapotranspiration

at the surface.

6. Coupling a ground heat exchanger model with a zone heat balance calculation can simultaneously predict the heat transfer rate between the zone, ground, and fluid. This alleviates the iteration required to simulate this condition using multiple decoupled models as in previous studies (Cullin et al., 2012).

5.3 Future Work

The following sections describe possible paths for further research studies in each of the three core topics of this work.

5.3.1 EnergyPlus Central Plant Simulation

The EnergyPlus central plant simulations were improved with a new solution algorithm. With this change in place, there is an opportunity for pursuing improved performance of advanced configurations.

5.3.1.1 Simulation Order

The dependence between simulation loops and half-loops is not easily identified for the generalized topology and coupling available in the improved solution algorithm. Although simple dependencies can be inferred based on component types and connections, a more robust and widely applicable method should be investigated. One possibility is to use graph theory to create a map of the simulation model, and find an optimal set of paths (simulation order). This alone could have the benefit of minimizing computation time, which is significant at this nested point within the whole building energy simulation environment. An advanced approach could also identify independent simulation paths, and simulate these paths concurrently, providing a further benefit. This would require much analysis, as threading this procedure introduces the possibility of multiple processes accessing and modifying shared data.

5.3.1.2 Central Plant Design

The central plant simulation model created in this work utilizes an algorithmic (non pressure-based) flow-solution to provide a suitable prediction of energy use for central plants. This limits the possibility to perform design calculations, such as optimal pipe diameter in a piping system. Further research could extend the model, including the pressure calculations, or implement a full flow network solution to provide capabilities such as this. Creating a piping system design focus within the context of a whole building energy simulation program would be a novel and useful research task.

5.3.1.3 Unifying Solution Algorithms

The level of integration between simulation systems inside the whole building energy simulation program EnergyPlus could be improved. The air system and hydronic system, while performing similar tasks and sharing similar topology rules, utilize two completely different simulation models. Coupling these could improve not only developer maintenance burden, but drastically improve the ability to advance the program by unifying the simulation methodology.

5.3.2 Transport Delay

The transport delay work included both experimental and modeling aspects. There are advancements available for improving delay modeling capabilities.

5.3.2.1 A Blended Flow Model

One result of the transport delay study in this work demonstrates that a blended flow model (using results from multiple models together) provides suitable results against one experimental data set. Further modeling work could include a robust, widely applicable, blended flow regime model, somewhere between well-mixed and plug-flow. One issue that may be difficult is describing system-specific attributes that induce mixing, though with a large study this could be evaluated into the weighting factors between methods.

5.3.2.2 Evaluating Immediate Temperature Response Effects

Transport delay experimental results were found to be in between the plug flow and well-mixed models, with the temperature response of each model differing significantly, especially during the initial start-up phase. At long time-scales, these results tend to become blurred into an overall energy impact, however if accuracy in the initial time is of importance, the selection of transport delay model is important. Quantifying these effects on specific applications where the initial time is important would suit as a future study.

5.3.2.3 Further Investigation on Optimal Node Discretization

The Hanby et al. (2002) model was described in section 3.2.4, with an emphasis on an issue related to the assertion of a single optimal node count value. This could be investigated further by setting up simulation models using experimental data measured during this study. At each experimental configuration, the discretization and time step could vary to determine if a different optimal point is attained, and if this optimal point could be predicted.

5.3.3 Ground Heat Exchanger Model

The ground heat exchanger model developed here coupled the ground with the zone heat balance calculations, and with the hydronic system simulation algorithms. High quality results were obtained, even with a coarse grid and low computational burden. Further work could be performed to improve the model results further and also to improve the usability of the model into different applications.

5.3.3.1 Investigating Far-field Boundary Specification

During the estimation of ground properties to be used in model calculations, the model was never able to fully match the undisturbed ground temperature at all measured depths. It is possible that this is because the far-field temperature correlation utilized did not fully capture the ground temperature variation. Future work could include implementing a far-field ground temperature boundary condition that captures more phenomena.

5.3.3.2 Corner Effects

The ground heat exchanger model developed here uses a uniform cross section, which is extended axially in the domain. This limits the applicability of the model, and further additions could include either:

- development of an approximation to be employed in the model that captures "corner" and other geometric variation effects, or
- modification of the model to actually include diverse geometries

However, additions such as these could result in actually reducing the model usability due to the increased computational and input burdens.

5.3.3.3 Generalized Applications

The generalized placement of pipes in the domain, along with the flexible integration characteristics, make this model especially viable for implementing capabilities for simulation additional heat exchanger applications.

Earth Tube An earth tube (Lee and Strand, 2008) is a duct buried in the ground which is used to pre-treat outside air being supplied to a zone. This is basically no different from a buried pipe with water or other fluid. This is an easily identified possibility for direct use of the model, as it would only require a change in working fluid and convection correlation.

Vertical Ground Heat Exchanger Vertical boreholes are used to interact predominantly with the deep ground nearly constant conditions. Over the length of the borehole the conditions will vary, as the shallow section encounters some effects from the ground surface, and the deeper sections encounter the thermal gradient in the ground and the possibility of end effects at the bottom of the tube. These are commonly used in groups of heat exchangers, not a single heat exchanger. During the final phase of this research work, a borehole model was implemented. To complete a vertical borehole field simulation using this ground heat exchanger model, the boundary condition specification must be adjusted to allow for the change in orientation and also to capture the vertical variation in boundary temperature and thermal properties.

References

Adjali, M., Davies, M. and Rees, S. (2004), 'A comparative study of design guide calculations and measured heat loss through the ground', *Building and Environment* **39**(11), 1301–1311.

Allen, R. G., Pereira, L. S., Raes, D. and Smith, M. (1998), *Crop Evapotranspiration - Guidelines for computing crop water requirements*, Food and Agriculture Organization of the United Nations.

Andolsun, S., Culp, C. H. and Haberl, J. (2010), EnergyPlus vs DOE-2: The effect of ground coupling on heating and cooling energy consumption of a slab-on-grade code house in a cold climate, *in* 'Proceedings of Simbuild 2010'.

Arakawa, M., Kagawa, T., Takeuchi, M., Rokuka, T. and Someya, T. (2008), A study on effects of mixing different gases in pipes, *in* '2008 Asia Simulation Conference - 7th International Conference on System Simulation and Scientific Computing, ICSC 2008', Institute of Electrical and Electronics Engineers, pp. 1046–1049.

Austin, W., Yavuzturk, C. and Spitler, J. D. (2000), 'Development of an in-situ system and analysis procedure for measuring ground thermal properties', *ASHRAE Transactions* **106**(1), 365–379.

Bahel, V., Said, S. and Abdelrahmen, M. (1989), 'Validation of DOE-2.1A and a microcomputer based hourly energy analysis computer program for a residential building', *Energy* **14**(4), 215–221.

Barbour, J. P. and Hittle, D. C. (2006), 'Modeling phase change materials with conduction transfer functions for passive solar applications', *Journal of Solar Energy Engineering* **128**(1), 58–68.

Bau, H. H. and Sadhai, S. S. (1982), 'Heat losses from a fluid flowing in a buried pipe', *International Journal of Heat and Mass Transfer* **25**(11), 1621–1629.

Beier, R. A., Smith, M. D. and Spitler, J. D. (2011), 'Reference data sets for vertical borehole ground heat exchanger models and thermal response test analysis', *Geothermics* **40**(1), 79–85.

Berghout, B. and Kuczera, G. (1994), Application of network linear programming to the pipe network problem, *in* 'National Conference Publication - Institution of Engineers, Australia', pp. 141–148.

Binks, J. (2011), Closing the loop; office tower simulation assumptions vs reality, *in* 'Proceedings of Building Simulation 2011'.

Blast Support Office (1986), 'Blast 3.0 users manual', [Software User's Manual].

Boulos, P. F. and Altman, T. (1993), 'An explicit approach for modelling closed pipes in water networks', *Applied Mathematical Modelling* **17**(8), 437–443.

Boulos, P. F. and Wood, D. J. (1990), 'Explicit calculation of pipe-network parameters', *Journal of Hydraulic Engineering* **116**(11), 1329–1344.

Braun, J. E. (1992), 'Comparison of chiller-priority storage-priority, and optimal control of an ice-storage system', *ASHRAE Transactions* **98**(1), 893–902.

Braun, J. E. (2007), 'Performance of a demand-limiting control algorithm for hybrid cooling plants.', *ASHRAE Transactions* **113**(2), 95–104.

Brazkeikis, L. (2010), 'Modeling and estimation of temperature transient process of heat carrier in heating system', *Elektronika ir elektrotechnika* **7**, 21–24.

Bronfenbrener, L. and Korin, E. (1999), 'Thawing and refreezing around a buried pipe', *Chemical Engineering and Processing* **38**(3), 239–247.

Buhl, W. F., Erdem, A. E., Eto, J. H., Hirsch, J. J. and Winkelmann, F. C. (1985), New features of the doe-2.1c energy analysis program, *in* 'International Conference Proceedings: IBPSA Building Simulation'.

Carslaw, H. S. and Jaeger, J. C. (1947), *Conduction of Heat in Solids*, Oxford University Press.

Chen, Q., Xu, C., Claridge, D., Turner, D. and Deng, S. (2007), Plant optimization program (POP) and its application in rate model for a large district energy and combined heat and power system, *in* 'International Conference Proceedings: IBPSA Building Simulation'.

Chengju, H., Changsheng, G. and Kai, X. (2012), 'Random heat temperature field model analysis on buried pipe of ground source heat pump', *Advanced Materials Research* **383**, 6626–6631.

Chung, M., Jung, P.-S. and Rangel, R. H. (1999), 'Semi-analytical solution for heat transfer from a buried pipe with convection on the exposed surface', *International Journal of Heat and Mass Transfer* **42**(20), 3771–3786.

Claesson, J. and Hagentoft, C.-E. (1991), 'Heat loss to the ground from a building - i. general theory', *Building and Environment* **26**(2), 195–208.

Claesson, J. and Hellstrom, G. (2011), 'Multipole method to calculate borehole thermal resistances in a borehole heat exchanger', *HVAC&R Research* **17**(6), 895–911.

Clark, D., Hurley, C. and Hill, C. (1985), 'Dynamic models for hvac system components', *ASHRAE Transactions* **91**(pt 1B), 737–751.

Collins, M. A. (1980), 'Pitfalls in pipe network analysis techniques', *Transportation Engineering Journal of ASCE* **106**(5), 507–521.

Crawley, D. B., Lawrie, L. K., Winkelmann, F. C., Buhl, W. F., Huang, Y. J., Pedersen, C. O., Strand, R. K., Liesen, R. J., Fisher, D. E., Witte, M. J. and Glazer, J. (2001), 'EnergyPlus: creating a new-generation building energy simulation program', *Energy and Buildings* **33**(4), 319–331.

Cross, H. (1936), Analysis of flow in networks of conduits or conductors, Technical Report Bulletin #286, University of Illinois Engineering Experimental Station.

Cullin, J. R., Spitler, J. D., Xing, L., Fisher, D. E. and Lee, E. S. (2012), 'Feasibility of foundation heat exchangers for residential ground source heat pump systems in the united states', *ASHRAE Transactions* **118**(1), 1039–1048.

Den Braven, K. R. and Nielsen, E. (1998), 'Performance prediction of a sub-slab heat exchanger for geothermal heat pumps', *Journal of Solar Energy Engineering* **120**(4), 282–288.

Dittus, P. W. and Boelter, L. M. (1930), 'Heat transfer in automobile radiators of the tubular type', *University of California Publications on Engineering* **2**, 443.

Ekambara, K. and Joshi, J. (2003), 'Axial mixing in pipe flows: turbulent and transition regions', *Chemical Engineering Science* **58**(12), 2715–2724.

Esen, H., Inalli, M. and Esen, M. (2007), 'Numerical and experimental analysis of a horizontal ground-coupled heat pump system', *Building and Environment* **42**(3), 1126–1134.

Estrada, C., Gonzalez, C., Aliod, R. and Pano, J. (2009), 'Improved pressurized pipe network hydraulic solver for applications in irrigation systems.', *Journal of Irrigation & Drainage Engineering* **135**(4), 421–430.

Fisher, D. E., Taylor, R. D., Buhl, F., Liesen, R. J. and Strand, R. K. (1999*a*), A modular, loop-based approach to hvac energy simulation and its implementation in energyplus, *in* 'Proceedings of Building Simulation '99', Vol. 3, pp. 1245–1252.

Fisher, D., Taylor, R., Buhl, F., Liesen, R. and Strand, R. (1999*b*), A modular, loop-based approach to hvac energy simulation and its implementation in energyplus, *in* 'Proceedings of Building Simulation '99', Vol. 3, pp. 1245–1252.

Fujieda, M. and Ohyama, Y. (1985), 'Analysis of mixture transport delay in fuel supply for carbureted engine', *Bulletin of the JSME* **28**(243), 2034–2040.

Gay, B. and Middleton, P. (1971), 'The solution of pipe network problems', *Chemical Engineering Science* **26**(1), 109–123.

Haghighi, K., Mohtar, R., Bralts, V. F. and Segerlind, L. J. (1992), 'A linear formulation model for pipe network components', *Computers and Electronics in Agriculture* **7**(4), 301–321.

HaktanIr, T. and ArdIclIoglu, M. (2004), 'Numerical modeling of darcy-weisbach friction factor and branching pipes problem', *Advances in Engineering Software* **35**(12), 773–779.

Hanby, V., Wright, J., Fletcher, D. and Jones, D. (2002), 'Modeling the dynamic response of conduits', *HVAC&R research* **8**(1), 1–12.

Hearns, G. and Grimble, M. (2010), Temperature control in transport delay systems, *in* 'Proceedings of the 2010 American Control Conference', Baltimore, MD, United states, pp. 6089–6094.

Higson, K. R. (1984), 'Pipe network analysis - linear solution for nodal heads', *Proceedings - Canadian Society for Civil Engineering* **2**, 977–988.

Hodge, B. K. (2006), 'A unified approach to piping system problems', *Computers in Education Journal* **16**(2), 68–79.

Holman, J. P. and Gajda, W. J., J. (1984), *Experimental Methods for Engineers*, McGraw-Hill Book Company.

Huang, Y. J. and Kung, H. L. (2005), 'Critical stability condition for ehss with friction and transport delay', *Journal of Dynamic Systems, Measurement and Control, Transactions of the ASME* **127**(2), 257–266.

Hunn, B. (1979), The DOE-2 building energy analysis computer program, *in* 'Proceedings of the Conference on conservation: energy management by design', El Paso, TX, pp. 182–187.

Hydeman, M., Taylor, S. and Winiarski, D. (2002), 'Application of component models for standards development', *ASHRAE Transactions* **108**(1), 742–750.

Ie, S., Itou, H., Iwano, R., Itou, H. and Matsui, S. (2001), Development of a simulation tool for pumping systems with separate simulators for piping and control subsystems, *in* '27th Annual Conference of the IEEE Industrial Electronics Society IECON 2001'.

Ihm, P. and Krarti, M. (2004), Implementation of two-dimensional foundation model for radiant floors into energyplus, *in* 'Proceedings of Simbuild 2004'.

Incropera, F. P. and DeWitt, D. P. (2002), *Introduction to Heat Transfer*, John Wiley & Sons.

Ingersoll, L. R. and Plass, H. J. (1948), 'Theory of the ground pipe source for the heat pump', *ASHVE Transactions* **54**, 339–348.

Jin, H. (2002), Parameter Estimation Based Models of Water Source Heat Pumps, PhD thesis, Oklahoma State University.

Johnston, S. P., Kazmer, D. O. and Gao, R. X. (2009), 'Online simulation-based process control for injection molding', *Polymer Engineering & Science* **49**(12), 2482–2491.

King, D. and Potter, Jr., R. (1998), 'Description of a steady-state cooling plant model developed for use in evaluating optimal control of ice thermal energy storage systems', *ASHRAE Transactions* **104**(1), 42–53.

Kohlenberg, M. and Wood, R. (1994), 'Pressure and flow transient response prediction for power plant piping networks', *Simulation* **63**(4), 235–248.

Kusuda, T. and Achenbach, P. (1965), Earth temperature and thermal diffusivity at selected stations in the united states, Technical report, National Bureau of Standards.

Lamberg, P., Lehtiniemi, R. and Henell, A. M. (2004), 'Numerical and experimental investigation of melting and freezing processes in phase change material storage', *International Journal of Thermal Sciences* **43**(3), 277–287.

Lang, F. D. and Miller, B. L. (1981), 'Use of friction-factor correlation in pipe-network problems', *Chemical Engineering* **88**(13), 95–97.

Lee, E. S., Fisher, D. E. and Spitler, J. D. (2013), 'Efficient horizontal ground heat exchanger simulation with zone heat balance integration', *HVAC&R Research.* **19**(3), 307–323.

Lee, K. H. and Strand, R. K. (2008), 'The cooling and heating potential of an earth tube system in buildings', *Energy and Buildings* **40**(4), 486–494.

Lee, S. and Prabhu, V. (2010), Simulation-based control for green transportation with high delivery service, *in* 'Proceedings of the 2010 Winter Simulation Conference', pp. 2046–2056.

Leung, K. S., Brill, E. D. and Rahman, M. S. (2000), Optimization of pipe sizing in underground water transmission network systems, *in* 'Proceedings of the ASCE National Conference on Environmental and Pipeline Engineering 2000', American Society of Civil Engineers, pp. 579–588.

Levenspiel, O. (1958), 'Longitudinal mixing of fluids flowing in circular pipes', *Industrial Engineering Chemistry Research* **50**(3), 343–346.

Liu, H., Yuan, Y., Zhao, M., Zheng, X., Lu, J. and Zhao, H. (2011), Study of mixing at cross junction in water distribution systems based on computational fluid dynamics, *in* 'International Conference on Pipelines and Trenchless Technology 2011: Sustainable Solutions for Water, Sewer, Gas, and Oil Pipelines, ICPTT 2011', American Society of Civil Engineers, Beijing, China, pp. 552–561.

Lopes, A. (2004), 'Implementation of the hardy-cross method for the solution of piping networks', *Computer Applications in Engineering Education* **12**(2), 117–125.

Mei, V. (1988), 'Heat pump ground coil analysis with thermal interference', *Journal of Solar Energy Engineering, Transactions of the ASME* **110**(2), 67–73.

Miles, S. (1975), 'Simple transport delay', *Electronic Engineering* **47**(567), 53–55.

Modelica Association (2010), *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification.*

Mohtar, R. H., Bralts, V. F. and Shayya, W. H. (1991), 'A finite element model for the analysis and optimization of pipe networks', *Transactions of the ASABE* **34**(2), 393–401.

Mosca, E. and Zappa, G. (1984), Mv adaptive controller for plants with time-varying i/o transport delay, *in* 'Proceedings of the IFAC Workshop', San Francisco, CA, USA, pp. 207–211.

Nauman, E. B. and Buffham, B. A. (1983), *Mixing in Continuous Flow Systems*, John Wiley & Sons.

Nazeer, M. M., Afzal, M., Tariq, G. F. and Ahmed, N. (1999), 'Global solution algorithm with some assisting techniques for modeling of unsteady flow in centrifuge cascade pipe network', *Computer Methods in Applied Mechanics and Engineering* **173**(1-2), 257–269.

Nelson, K. P. (1999), 'Dynamics of primary/secondary chilled water systems', *ASHRAE Transactions* **105**(2), 1–7.

Ngo, C. C. and Lai, F. C. (2005), 'Effects of backfill on heat transfer from a buried pipe', *Journal of Heat Transfer* **127**(7), 780–784.

Nielsen, H. B. (1989), 'Methods for analyzing pipe networks', *Journal of Hydraulic Engineering* **115**(2), 139–157.

Oke, I. A. (2007), 'Reliability and statistical assessment of methods for pipe network analysis', *Environmental Engineering Science* **24**(10), 1481–1490.

Pasqualetto, L., Zmeureanu, R. and Fazio, P. (1998), 'A case study of validation of an energy analysis program: MICRO-DOE2.1E', *Building and Environment* **33**(1), 21–41.

Phalak, K. (2011), Development, implementation and validation of a non-dimensional pump model in energyplus, Master's thesis, Oklahoma State University.

Piechowski, M. (1996), A Ground Coupled Heat Pump system with Energy Storage, PhD thesis, Melbourne University.

Piechowski, M. (1999), 'Heat and mass transfer model of a ground heat exchanger: Theoretical development', *International Journal of Energy research* **23**(7), 571–588.

Preece, P. E. and Ti, H. C. (1989), 'Matrix method for the solution of fluid network problems: Mixed pressure and flow specification', *The Chemical Engineering Journal* **40**(1), 1–6.

Raymond, J., Therrien, R., Gosselin, L. and Lefebvre, R. (2011), 'Numerical analysis of thermal response tests with a groundwater flow and heat transfer model', *Renewable Energy* **36**(1), 315–324.

Sadegh, A. M., Jiji, L. M. and Weinbaum, S. (1987), 'Boundary integral equation technique with application to freezing around a buried pipe', *International Journal of Heat and Mass Transfer* **30**(2), 223–232.

Said, S. M., Habib, M. A., Mokheimer, E., Al-Shayea, N. and Sharqawi, M. (2009), Horizontal ground heat exchanger design for ground-coupled heat pumps, *in* 'Proceedings of EVER Monaco 2009'.

Sakamoto, Y., Nagaiwa, A., Kobayasi, S. and Shinozaki, T. (1999), 'Optimization method of district heating and cooling plant operation based on genetic algorithm', *ASHRAE Transactions* **105**, 104–115.

Sekhar, S. and Yat, C. J. (1998), 'Energy simulation approach to air-conditioning system evaluation', *Building and Environment* **33**(6), 397–408.

Speetjens, M., Metcalfe, G. and Rudman, M. (2006), 'Topological mixing study of non-newtonian duct flows', *Physics of Fluids* **18**(10), 1–11.

Spitler, J., Xing, L., Cullin, J., Fisher, D., Shonder, J. and Im, P. (2010), Residential ground source heat pump systems utilizing foundation heat exchangers, *in* 'Proceedings of Clima 2010', Antalya Turkey.

Stocki, M., Curcija, D. C. and Bhandari, M. S. (2007), 'The development of standardized whole-building simulation assumptions for energy analysis for a set of commercial buildings.', *ASHRAE Transactions* **113**(2), 422–435.

Sun, J. (2010), 'Optimal supervisory control of a central chilled water plant with heuristic search sequential quadratic programming', *Engineering Optimization* **42**(9), 863–885.

Tamir, A. and Taitel, Y. (1972), 'On the concept of the "mixing cup" temperature in flows with axial conduction', *The Canadian Journal of Chemical Engineering* **50**(3), 421–424.

Taylor, R. D., Pedersen, C., Liesen, R., Fisher, D. and Lawrie, L. (1991), Impact of simultaneous simulation of buildings and mechanical systems in heat balance based energy analysis programs on system response and control, *in* '2nd International Conference Proceedings: IBPSA Building Simulation '91', Nice, France, pp. 227–234.

Thomas, H. R. and Rees, S. W. (2009), 'Measured and simulated heat transfer to foundation soils', *Geotechnique* **59**(4), 365–375.

Tian, Z., Love, J. A. and Tian, W. (2009), 'Applying quality control in building energy modelling: comparative simulation of a high performance building', *Journal of Building Performance Simulation* **2**(3), 163–178.

Tobias, J. R. (1973), 'Simplified transfer function for temperature response of fluids flowing through coils, pipes or ducts', *ASHRAE Transactions* **79**(2), 19–22.

Todini, E. (2006), On the convergence properties of the different pipe network algorithms, *in* '8th Annual Water Distribution Systems Analysis Symposium 2006'.

United States Department of Energy (2012), *EnergyPlus: Input-Output Reference*.

van Zyl, J. E., Kumar, P. and Gupta, M. (2008), 'Two-point linearization method for the analysis of pipe networks', *Journal of Hydraulic Engineering* **134**(8), 1176–1179.

Wang, F., Yoshida, H., Ono, E. and Shingu, H. (2007), Methodology for optimizing the operation of heating/cooling plants with multi-heat source equipments using simulation, *in* 'International Conference Proceedings: IBPSA Building Simulation'.

Webb, S. W. and Van Bloemen Waanders, B. G. (2007), High fidelity computational fluid dynamics for mixing in water distribution systems, *in* '8th Annual Water Distribution Systems Analysis Symposium 2006'.

Wetter, M. (2009), A modelica-based model library for building energy and control systems, *in* 'Eleventh International IBPSA Conference'.

Wilcox, S. and Marion, W. (2008), Users manual for tmy3 data sets, Technical Report NREL/TP-581-43156, National Renewable Energy Laboratory.

Witte, M. J., Pedersen, C. O. and Spitler, J. D. (1989), Techniques for simulataneous simulation of buildings and mechanical systems in heat balance based energy analysis programs, *in* 'Proceedings of Building Simulation 1989'.

Wood, D. J. and Charles, C. O. A. (1972), 'Hydraulic network analysis using linear theory', *ASCE Journal of the Hydraulics Division* **98**(HY7), 1157–1170.

Xing, L., Cullin, J. R., Spitler, J. D., Im, P. and Fisher, D. E. (2011), 'Foundation heat exchangers for residential ground source heat pump systems–numerical modeling and experimental validation', *HVAC&R Research* **17**(6), 1059–1074.

Xu, X. (2007), Simulation and Optimal Control of Hybrid Ground Source Heat Pump Systems, PhD thesis, Oklahoma State University.

Yavuzturk, C. and Spitler, J. D. (1999), 'A short time step response factor model for vertical ground loop heat exchangers', *ASHRAE Transactions* **105**(2), 475–485.

Zhu, H., Zhu, G., Lu, W. and Zhang, Y. (2012), Optimal hydraulic design and numerical simulation of pumping systems, *in* ‘2012 International Conference on Modern Hydraulic Engineering’, pp. 75–80.

APPENDIX A

EnergyPlus Input Specification for Model Abstraction Case A

Listing A.1: Input Data File (IDF) Listing

```

Version,7.2;

Building,
  Plant Load Profile Example,  !- Name
  0.0,                        !- North Axis {deg}
  Suburbs,                    !- Terrain
  0.04,                       !- Loads Convergence Tolerance Value
  0.04,                       !- Temperature Convergence Tolerance Value {deltaC}
  FullInteriorAndExterior,    !- Solar Distribution
  25,                         !- Maximum Number of Warmup Days
  6;                          !- Minimum Number of Warmup Days

Timestep,6;

GlobalGeometryRules,
  UpperLeftCorner,            !- Starting Vertex Position
  CounterClockWise,           !- Vertex Entry Direction
  Relative;                   !- Coordinate System

Site:Location,
Tulsa,                        !- Location Name
    36.20,                    !- Latitude {N+ S-}
   -95.88,                    !- Longitude {W- E+}
    -6.00,                    !- Time Zone Relative to GMT {GMT+/-}
   198.00;                    !- Elevation {m}

RunPeriod,
  ,                           !- Name
  7,                          !- Begin Month
  1,                          !- Begin Day of Month
  7,                          !- End Month
  1,                          !- End Day of Month
  Tuesday,                    !- Day of Week for Start Day
  Yes,                        !- Use Weather File Holidays and Special Days
  Yes,                        !- Use Weather File Daylight Saving Period
  No,                         !- Apply Weekend Holiday Rule
  Yes,                        !- Use Weather File Rain Indicators
  Yes;                        !- Use Weather File Snow Indicators

SimulationControl,
  No,                         !- Do Zone Sizing Calculation
  No,                         !- Do System Sizing Calculation
  No,                         !- Do Plant Sizing Calculation
  No,                         !- Run Simulation for Sizing Periods
  Yes;                        !- Run Simulation for Weather File Run Periods

PlantLoop,
  Main Loop,                  !- Name
  WATER,                      !- Fluid Type
  ,                           !- User Defined Fluid Type
  Main Loop Operation,        !- Plant Equipment Operation Scheme Name
  Supply Outlet Node,         !- Loop Temperature Setpoint Node Name
  100,                        !- Maximum Loop Temperature {C}
  3,                          !- Minimum Loop Temperature {C}
  0.003,                      !- Maximum Loop Flow Rate {m3/s}
  0,                          !- Minimum Loop Flow Rate {m3/s}
  autocalculate,              !- Plant Loop Volume {m3}
  Supply Inlet Node,          !- Plant Side Inlet Node Name
  Supply Outlet Node,         !- Plant Side Outlet Node Name
  Supply Branches,            !- Plant Side Branch List Name
  Supply Connectors,          !- Plant Side Connector List Name
  Demand Inlet Node,          !- Demand Side Inlet Node Name
  Demand Outlet Node,         !- Demand Side Outlet Node Name
  Demand Branches,            !- Demand Side Branch List Name
  Demand Connectors,          !- Demand Side Connector List Name
  Sequential; !OPTIMAL;                   !- Load Distribution Scheme

SetpointManager:Scheduled,
  Main Loop Setpoint Manager,  !- Name
  Temperature,                !- Control Variable
  Main Loop Temp Sch,         !- Schedule Name
  Main Loop Setpoint Node List;  !- Setpoint Node or NodeList Name
```

```

NodeList,
  Main Loop Setpoint Node List,  !- Name
  Supply Outlet Node,      !- Node 1 Name
  Chiller1 Outlet Node,    !- Node 2 Name
  Chiller2 Outlet Node;    !- Node 3 Name

PlantEquipmentOperationSchemes,
  Main Loop Operation,     !- Name
  PlantEquipmentOperation:ComponentSetpoint,  !- Control Scheme 1 Object Type
  ComponentSetpointChillers,          !- Control Scheme 1 Name
  AlwaysOnSchedule;        !- Control Scheme 1 Schedule Name

PlantEquipmentOperation:ComponentSetpoint,
  ComponentSetpointChillers, !- Name
  Chiller:ConstantCOP,       !- Equipment 1 Object Type
  Chiller1,                  !- Equipment 1 Name
  Chiller1 Inlet Node,         !- Demand Calculation 1 Node Name
  Chiller1 Outlet Node,        !- Setpoint 1 Node Name
  0.0015,                    !- Component 1 Flow Rate
  Cooling,                   !- Operation 1 Type
  Chiller:ConstantCOP,       !- Equipment 2 Object Type
  Chiller2,                  !- Equipment 2 Name
  Chiller2 Inlet Node,         !- Demand Calculation 2 Node Name
  Chiller2 Outlet Node,        !- Setpoint 2 Node Name
  0.0015,                    !- Component 2 Flow Rate
  Cooling;                   !- Operation 2 Type

PlantEquipmentList,
  Chiller Plant,           !- Name
  Chiller:ConstantCOP,       !- Equipment 1 Object Type
  Chiller1,                  !- Equipment 1 Name
  Chiller:ConstantCOP,       !- Equipment 1 Object Type
  Chiller2;                  !- Equipment 1 Name

BranchList,
  Supply Branches,         !- Name
  Supply Inlet Branch,     !- Branch 1 Name
  Chiller1 Branch,         !- Branch 2 Name
  Chiller2 Branch,         !- Branch 3 Name
  Supply Outlet Branch;    !- Branch 4 Name

ConnectorList,
  Supply Connectors,       !- Name
  Connector:Splitter,      !- Connector 1 Object Type
  Supply Splitter,         !- Connector 1 Name
  Connector:Mixer,         !- Connector 2 Object Type
  Supply Mixer;            !- Connector 2 Name

Connector:Splitter,
  Supply Splitter,         !- Name
  Supply Inlet Branch,     !- Inlet Branch Name
  Chiller1 Branch,         !- Branch 2 Name
  Chiller2 Branch;         !- Branch 3 Name

Connector:Mixer,
  Supply Mixer,            !- Name
  Supply Outlet Branch,    !- Outlet Branch Name
  Chiller1 Branch,         !- Branch 2 Name
  Chiller2 Branch;         !- Branch 3 Name

Branch,
  Supply Inlet Branch,     !- Name
  0,                       !- Maximum Flow Rate {m3/s}
  ,                        !- Pressure Drop Curve Name
  Pipe:Adiabatic,          !- Component 1 Object Type
  Supply Inlet Pipe,       !- Component 1 Name
  Supply Inlet Node,       !- Component 1 Inlet Node Name
  Supply Inlet Pipe Outlet Node, !- Component 1 Outlet Node Name
  PASSIVE;                 !- Component 1 Branch Control Type

Pipe:Adiabatic,
  Supply Inlet Pipe,       !- Name
  Supply Inlet Node,       !- Inlet Node Name
  Supply Inlet Pipe Outlet Node; !- Outlet Node Name

Branch,
  Chiller1 Branch,         !- Name
  0,                       !- Maximum Flow Rate {m3/s}
  ,                        !- Pressure Drop Curve Name
  Pump:ConstantSpeed,      !- Component 1 Object Type
  Pump1,                   !- Component 1 Name
  Pump1 Inlet Node,        !- Component 1 Inlet Node Name
  Chiller1 Inlet Node,     !- Component 1 Outlet Node Name
  SeriesActive,            !- Component 1 Branch Control Type
  Chiller:ConstantCOP,      !- Component 1 Object Type
  Chiller1,                   !- Component 1 Name
  Chiller1 Inlet Node,        !- Component 1 Inlet Node Name
  Chiller1 Outlet Node,     !- Component 1 Outlet Node Name
  SeriesActive;            !- Component 1 Branch Control Type

Pump:ConstantSpeed,
  Pump1,                   !- Name
  Pump1 Inlet Node,         !- Inlet Node Name
  Chiller1 Inlet Node,      !- Outlet Node Name
  0.0015,                   !- Rated Flow Rate
  10000,                    !- Rated Pump Head
  25,                       !- Rated Power Consumption
  ,                         !- Motor Efficiency
  ,                         !- Fraction of Motor Inefficiencies to Fluid Stream
  Intermittent;             !- Pump Control Type

Chiller:ConstantCOP,
  Chiller1,                 !- Name
  20000,                    !- Nominal Capacity
  3.5,                      !- Nominal COP
  0.0015,                   !- Design Chilled Water Flow Rate
  ,                         !- Design Condenser Water Flow Rate
  Chiller1 Inlet Node,      !- Chilled Water Inlet Node Name
  Chiller1 Outlet Node,     !- Chilled Water Outlet Node Name
  Chiller2 Condenser Inlet Node, !- Condenser Inlet Node Name
  ,                         !- Condenser Outlet Node Name
  AirCooled,                !- Condenser Type
  VariableFlow,             !- Chiller Flow Mode
  1.0;                      !- Sizing Factor

OutdoorAir:Node,Chiller1 Condenser Inlet Node;

Branch,
  Chiller2 Branch,          !- Name
  0,                        !- Maximum Flow Rate {m3/s}
  ,                         !- Pressure Drop Curve Name
  Pump:ConstantSpeed,       !- Component 1 Object Type
  Pump2,                    !- Component 1 Name
  Pump2 Inlet Node,         !- Component 1 Inlet Node Name
  Chiller2 Inlet Node,      !- Component 1 Outlet Node Name
  SeriesActive,             !- Component 1 Branch Control Type
  Chiller:ConstantCOP,       !- Component 1 Object Type
  Chiller2,                    !- Component 1 Name
  Chiller2 Inlet Node,         !- Component 1 Inlet Node Name
  Chiller2 Outlet Node,      !- Component 1 Outlet Node Name
  SeriesActive;             !- Component 1 Branch Control Type

Pump:ConstantSpeed,
  Pump2,                    !- Name
  Pump2 Inlet Node,         !- Inlet Node Name
  Chiller2 Inlet Node,      !- Outlet Node Name
  0.0015,                   !- Rated Flow Rate
  10000,                    !- Rated Pump Head
  25,                       !- Rated Power Consumption
  ,                         !- Motor Efficiency
  ,                         !- Fraction of Motor Inefficiencies to Fluid Stream
  Intermittent;             !- Pump Control Type

Chiller:ConstantCOP,
  Chiller2,                 !- Name
  20000,                    !- Nominal Capacity
  3.5,                      !- Nominal COP
  0.0015,                   !- Design Chilled Water Flow Rate
  ,                         !- Design Condenser Water Flow Rate
  Chiller2 Inlet Node,      !- Chilled Water Inlet Node Name
  Chiller2 Outlet Node,     !- Chilled Water Outlet Node Name
  Chiller2 Condenser Inlet Node, !- Condenser Inlet Node Name
  ,                         !- Condenser Outlet Node Name
  AirCooled,                !- Condenser Type
  VariableFlow,             !- Chiller Flow Mode
  1.0;                      !- Sizing Factor

OutdoorAir:Node,Chiller2 Condenser Inlet Node;

Branch,
  Supply Outlet Branch,     !- Name
  0,                        !- Maximum Flow Rate {m3/s}
  ,                         !- Pressure Drop Curve Name
  Pipe:Adiabatic,           !- Component 1 Object Type
  Supply Outlet Pipe,       !- Component 1 Name
  Supply Outlet Pipe Inlet Node, !- Component 1 Inlet Node Name
  Supply Outlet Node,       !- Component 1 Outlet Node Name
  PASSIVE;                  !- Component 1 Branch Control Type

Pipe:Adiabatic,
  Supply Outlet Pipe,       !- Name
  Supply Outlet Pipe Inlet Node,!- Inlet Node Name
  Supply Outlet Node;       !- Outlet Node Name

BranchList,
  Demand Branches,          !- Name
  Demand Inlet Branch,      !- Branch 1 Name
  Load Profile Branch 1,    !- Branch 2 Name
  Load Profile Branch 2,    !- Branch 3 Name
  Bypass Branch,            !- Branch 4 Name
  Demand Outlet Branch;     !- Branch 5 Name

ConnectorList,
  Demand Connectors,        !- Name
  Connector:Splitter,       !- Connector 1 Object Type
  Demand Splitter,         !- Connector 1 Name
  Connector:Mixer,         !- Connector 2 Object Type
  Demand Mixer;            !- Connector 2 Name

Connector:Splitter,
  Demand Splitter,         !- Name
  Demand Inlet Branch,     !- Inlet Branch Name
  Load Profile Branch 1,   !- Outlet Branch 1 Name
  Load Profile Branch 2,   !- Outlet Branch 2 Name
  Bypass Branch;           !- Outlet Branch 3 Name

Connector:Mixer,
  Demand Mixer,            !- Name
  Demand Outlet Branch,    !- Outlet Branch Name
  Load Profile Branch 1,   !- Inlet Branch 1 Name
  Load Profile Branch 2,   !- Inlet Branch 2 Name
  Bypass Branch;           !- Inlet Branch 3 Name

Branch,
  Demand Inlet Branch,     !- Name
  0,                       !- Maximum Flow Rate {m3/s}
  ,                        !- Pressure Drop Curve Name
  Pipe:Adiabatic,          !- Component 1 Object Type
  Demand Inlet Pipe,       !- Component 1 Name
  Demand Inlet Node,       !- Component 1 Inlet Node Name
  Demand Pipe-Load Profile Node,  !- Component 1 Outlet Node Name
  PASSIVE;                 !- Component 1 Branch Control Type

Pipe:Adiabatic,
  Demand Inlet Pipe,       !- Name
  Demand Inlet Node,       !- Inlet Node Name
  Demand Pipe-Load Profile Node;  !- Outlet Node Name

Branch,
  Load Profile Branch 1,   !- Name
  0,                       !- Maximum Flow Rate {m3/s}
  ,                        !- Pressure Drop Curve Name
  LoadProfile:Plant,       !- Component 1 Object Type
  Load Profile 1,          !- Component 1 Name
  Demand Load Profile 1 Inlet Node,  !- Component 1 Inlet Node Name
  Demand Load Profile 1 Outlet Node,  !- Component 1 Outlet Node Name
  ACTIVE;                  !- Component 1 Branch Control Type

LoadProfile:Plant,
  Load Profile 1,          !- Name
  Demand Load Profile 1 Inlet Node,  !- Inlet Node Name
  Demand Load Profile 1 Outlet Node,  !- Outlet Node Name
  Load Profile 1 Load Schedule,  !- Load Schedule Name
  0.0015,                   !- Peak Flow Rate {m3/s}
  Load Profile 1 Flow Frac Schedule;  !- Flow Rate Fraction Schedule Name

Schedule:Compact,
  Load Profile 1 Load Schedule,  !- Name
  Any Number,              !- Schedule Type Limits Name
  THROUGH: 12/31,          !- Field 1
  FOR: AllDays,            !- Field 2
  UNTIL: 7:00,0.0,        !- Field 3
  UNTIL: 9:00,-6000,        !- Field 5
  UNTIL: 12:00,-10000,           !- Field 7
  UNTIL: 18:00,-16000,       !- Field 9
  UNTIL: 20:00,-9000,      !- Field 11
  UNTIL: 24:00,0.0;      !- Field 13

Schedule:Compact,
  Load Profile 1 Flow Frac Schedule,  !- Name
  Any Number,              !- Schedule Type Limits Name
  THROUGH: 12/31,          !- Field 1
  FOR: AllDays,            !- Field 2
  UNTIL: 7:00,0.0,        !- Field 3
  UNTIL: 9:00,0.3,        !- Field 5
  UNTIL: 12:00,0.5,           !- Field 7
  UNTIL: 18:00,1.0,       !- Field 9
  UNTIL: 20:00,0.4,      !- Field 11
  UNTIL: 24:00,0.0;      !- Field 13

Branch,
  Load Profile Branch 2,   !- Name
  0,                       !- Maximum Flow Rate {m3/s}
  ,                        !- Pressure Drop Curve Name
  LoadProfile:Plant,       !- Component 1 Object Type
  Load Profile 2,          !- Component 1 Name
  Demand Load Profile 2 Inlet Node,  !- Component 1 Inlet Node Name
  Demand Load Profile 2 Outlet Node,  !- Component 1 Outlet Node Name
  ACTIVE;                  !- Component 1 Branch Control Type

LoadProfile:Plant,
  Load Profile 2,          !- Name
  Demand Load Profile 2 Inlet Node,  !- Inlet Node Name
  Demand Load Profile 2 Outlet Node,  !- Outlet Node Name
  Load Profile 2 Load Schedule,  !- Load Schedule Name
  0.0015,                   !- Peak Flow Rate {m3/s}
  Load Profile 2 Flow Frac Schedule;  !- Flow Rate Fraction Schedule Name

```

```
Schedule:Compact,
  Load Profile 2 Load Schedule,  !- Name
  Any Number,               !- Schedule Type Limits Name
  THROUGH: 12/31,           !- Field 1
  FOR: AllDays,             !- Field 2
  UNTIL: 7:00,0.0,         !- Field 3
  UNTIL: 9:00,0.0,         !- Field 5
  UNTIL: 12:00,-10000,            !- Field 7
  UNTIL: 18:00,-16000,        !- Field 9
  UNTIL: 20:00,-9000,       !- Field 11
  UNTIL: 24:00,0.0;       !- Field 13

Schedule:Compact,
  Load Profile 2 Flow Frac Schedule,  !- Name
  Any Number,               !- Schedule Type Limits Name
  THROUGH: 12/31,           !- Field 1
  FOR: AllDays,             !- Field 2
  UNTIL: 7:00,0.0,         !- Field 3
  UNTIL: 9:00,0.0,         !- Field 5
  UNTIL: 12:00,0.5,            !- Field 7
  UNTIL: 18:00,1.0,        !- Field 9
  UNTIL: 20:00,0.4,       !- Field 11
  UNTIL: 24:00,0.0;       !- Field 13

Branch,
  Bypass Branch,      !- Name
  0,                        !- Maximum Flow Rate {m3/s}
  ,                         !- Pressure Drop Curve Name
  Pipe:Adiabatic,           !- Component 1 Object Type
  Bypass Pipe,        !- Component 1 Name
  Bypass Inlet Node,        !- Component 1 Inlet Node Name
  Bypass Outlet Node,  !- Component 1 Outlet Node Name
  PASSIVE;                  !- Component 1 Branch Control Type

Pipe:Adiabatic,
  Bypass Pipe,        !- Component 1 Name
  Bypass Inlet Node,        !- Component 1 Inlet Node Name
  Bypass Outlet Node;  !- Component 1 Outlet Node Name

Branch,
  Demand Outlet Branch,     !- Name
  0,                        !- Maximum Flow Rate {m3/s}
  ,                         !- Pressure Drop Curve Name
  Pipe:Adiabatic,           !- Component 1 Object Type
  Demand Outlet Pipe,       !- Component 1 Name
  Demand Load Profile-Pipe Node,  !- Component 1 Inlet Node Name
  Demand Outlet Node,       !- Component 1 Outlet Node Name
  PASSIVE;                  !- Component 1 Branch Control Type

Pipe:Adiabatic,
  Demand Outlet Pipe,       !- Name
  Demand Load Profile-Pipe Node,  !- Inlet Node Name
  Demand Outlet Node;       !- Outlet Node Name

ScheduleTypeLimits,
  Any Number;               !- Name

ScheduleTypeLimits,
  On/Off,                   !- Name
  0,                        !- Lower Limit Value
  1,                        !- Upper Limit Value
  DISCRETE;                 !- Numeric Type

Schedule:Compact,
  Main Loop Temp Sch,       !- Name
  Any Number,               !- Schedule Type Limits Name
  THROUGH: 12/31,           !- Field 1
  FOR: AllDays,             !- Field 2
  UNTIL: 24:00,7.22;        !- Field 3

Schedule:Compact,
  AlwaysOnSchedule,         !- Name
  On/Off,                   !- Schedule Type Limits Name
  THROUGH: 12/31,           !- Field 1
  FOR: AllDays,             !- Field 2
  UNTIL: 24:00,1;           !- Field 3

Output:VariableDictionary,Regular;

Output:Variable,*,System Node Temp,Timestep;

Output:Variable,*,System Node MassFlowRate,Timestep;

Output:Variable,*,Chiller Evap Heat Trans Rate,Timestep;

Output:Variable,*,Schedule Value,Timestep;

Output:Variable,*,System Node MassFlowRateRequest,Timestep;

Output:Variable,*,Plant Load Profile Mass Flow Rate,Timestep;

Output:Variable,*,Plant Load Profile Heat Transfer Rate,Timestep;

```

```
Output:Variable,*,Plant Load Profile Heat Transfer Energy,Timestep;

Output:Meter:MeterFileOnly,Electricity:Facility,monthly;

Output:Meter:MeterFileOnly,Electricity:Plant,monthly;

Output:Meter:MeterFileOnly,Electricity:Facility,runperiod;

Output:Meter:MeterFileOnly,Electricity:Plant,runperiod;

Output:Diagnostics,DisplayAdvancedReportVariables;
```

APPENDIX B

Standalone Ground Heat Exchanger Model Source

Listing B.1: Standalone Ground Heat Exchanger Model Source: Manager

```
! This work authored by Edwin Lee, at Oklahoma State University
!  ___  _     ____  _        _
! / _ \| | __/ ___|| |_ __ _| |_ ___
!| | | | |/ /\___ \| __/ _` | __/ _ \
!| |_| |   <  ___) | || (_| | ||  __/
! \___/|_|\_\|____/ \__\__,_|\__\___|

PROGRAM RunPiechowski

    USE PiechowskiInputManager
    USE PiechowskiSimulationManager

    IMPLICIT NONE

    !Simulation parameters
    REAL(r64), PARAMETER :: TimeStepSize = 900.0d0
    INTEGER,   PARAMETER :: NumTimeSteps = 8760*4
    INTEGER,   PARAMETER :: OutputInterval = 250
    INTEGER,   PARAMETER :: FieldReportIncrement = 100
    LOGICAL,   PARAMETER :: DoingUGTValidation = .TRUE.
    LOGICAL,   PARAMETER :: WriteTempProfiles = .FALSE.

    !File unit numbers
    INTEGER, PARAMETER :: TempUnitNum = 29
    INTEGER, PARAMETER :: SimConditionsUnitNum = 30
    INTEGER, PARAMETER :: CircuitUnitNum = 31
    INTEGER, PARAMETER :: CircuitProfilesUnitNum = 32
    INTEGER, PARAMETER :: GroundTemperaturesUnitNum = 33

    !Flags, reporting values
    INTEGER :: TimeStepIndex
    INTEGER :: NumIterationsUsed
    LOGICAL :: ErrorsFound
    INTEGER, DIMENSION(8) :: DateTimeVals
    LOGICAL :: exists

    !Ground temperature (UGT Validation) variables
    REAL(r64), ALLOCATABLE, DIMENSION(:) :: GroundTempData !ground depths to be calculated for verification at
         depths 1' to 6'
    INTEGER :: NumTempsToReport
    INTEGER :: CellAverage_Start
    INTEGER :: CellAverage_Step
    INTEGER :: CalcInterval
    INTEGER :: XValue
    INTEGER :: ZValue
    INTEGER :: DepthIndex
    INTEGER :: SubIndex
    INTEGER :: SubSubIndex
    INTEGER :: SubSubCounter
    INTEGER :: StartingY
    INTEGER :: EndingY

    !Counter-indeces
    INTEGER :: Y, Z, I, X

    !Character formats
    CHARACTER(LEN=20) :: c_size        !x-direction cell count for easy format specification
    CHARACTER(LEN=20) :: fmt_domain    !format statement of c_size real values
    CHARACTER(LEN=20) :: fmt_domain2   !format statement of c_size integers
    CHARACTER(LEN=20) :: fmt_domain3   !format statement of c_size alpha strings
    CHARACTER(LEN=20) :: fmt_circuit   !similar to domain except # is the number of circuit cells
    CHARACTER(LEN=20) :: fmt_circuit2  !similar to domain except # is the number of circuit cells
    CHARACTER(LEN=60) :: GroundTemperaturesFormat
    CHARACTER(LEN=1000) :: GroundTemperaturesHeader
    CHARACTER(LEN=36) :: TimeStepFileName

    !Cell type strings for nicer reporting
    CHARACTER(len=6), PARAMETER, DIMENSION(-10:-1) :: celltypenames = (/'ba_cut', 'ba_cor', 'ba_flo', 'ba_wal', '
         bn_adb', 'bn_far', 'bn_sur', 'gfield', '_pipe_', '_????_'/)

    !Initial reporting
    WRITE(*, *) '***␣Multipipe␣***␣<Standlone␣Edition>␣***'
    CALL DATE_AND_TIME(VALUES=DateTimeVals)
```

```
WRITE(*, *) 'Starting␣Date␣and␣Time:'
WRITE(*, '(I4,␣"-",␣I2,␣"-",␣I2,␣"␣␣--␣␣",␣I2,␣":",␣I2,␣":",␣I2)') &
    DateTimeVals(1), DateTimeVals(2), DateTimeVals(3), DateTimeVals(5), DateTimeVals(6), DateTimeVals(7)
WRITE(*, *) 'Beginning␣Simulation'

!Process input and build mesh
CALL PerformInputProcessing()

!Reporting
WRITE(*, *) 'Completed␣Input␣Processing␣and␣Mesh␣Development'

!Set up some formatting now that we know mesh size
WRITE(c_size, '(I2)') SIZE(Cells,1)
WRITE(fmt_domain, *) "("//TRIM(c_size)//"(F7.3,"","",""))"
WRITE(fmt_domain2, *) "("//TRIM(c_size)//"(I7,"","",""))"
WRITE(fmt_domain3, *) "("//TRIM(c_size)//"(A6,"","",""))"

!Open output files
OPEN(UNIT=SimConditionsUnitNum, STATUS='REPLACE', FILE='MainOutput.csv')
IF (Has%PipeCircuit) OPEN(UNIT=CircuitProfilesUnitNum, STATUS='REPLACE', FILE='PipeProfiles.csv')
IF (DoingUGTValidation) OPEN(UNIT=GroundTemperaturesUnitNum, STATUS='REPLACE', FILE='GroundTemperatures.csv')

INQUIRE(FILE=stopFile, EXIST=exists)
IF (exists) CALL system('rm␣"' // trim(stopFile) // '"')

!Begin time step loop
DO TimeStepIndex = 1, NumTimeSteps

    !perform a simulation time step
    CALL PerformSimulation(TimeStepIndex, TimeStepSize, NumIterationsUsed, ErrorsFound)

    !do some one time reporting
    IF (TimeStepIndex == 1) THEN

        !Write the main output file header
        WRITE(SimConditionsUnitNum, '(A100)') &
               'TimeStepIndex,␣Num␣Iterations,␣Outdoor␣Dry␣Bulb,␣Relative␣Humidity,␣Wind␣Speed,␣Solar␣
                    Radiation'
        WRITE(SimConditionsUnitNum, '(A35)') &
               '[-],␣[-],␣[C],␣[%],␣[m/s],␣[W/m2]'

        !write a cartesian temperature profile after one time step is done
        OPEN(UNIT=TempUnitNum, STATUS='REPLACE', FILE='After1TimeStepTemps.csv')
        CALL FlushTemperatureField(TempUnitNum, fmt_domain)
        CLOSE(TempUnitNum)

        !and write the cell type strings
        OPEN(UNIT=TempUnitNum, STATUS='REPLACE', FILE='CellTypes.csv')
        DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
            WRITE(TempUnitNum, *) '␣'
            WRITE(TempUnitNum, '(I2)') Z
            DO Y = UBOUND(Cells,2), LBOUND(Cells,2), -1
                WRITE(TempUnitNum, fmt_domain3) (celltypenames(Cells(:, Y, Z)%CellType))
            END DO
        END DO
        CLOSE(TempUnitNum)

        !write some cell properties
        OPEN(UNIT=TempUnitNum, STATUS='REPLACE', FILE='CellGeometry.csv')
        WRITE(TempUnitNum, *) 'Cell␣Centroid␣X'
        DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
            WRITE(TempUnitNum, *) '␣'
            WRITE(TempUnitNum, '(I2)') Z
            DO Y = UBOUND(Cells,2), LBOUND(Cells,2), -1
                WRITE(TempUnitNum, fmt_domain) (Cells(:, Y, Z)%Centroid%X)
            END DO
        END DO
        WRITE(TempUnitNum, *) '␣'
        WRITE(TempUnitNum, *) 'Cell␣Centroid␣Y'
        DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
            WRITE(TempUnitNum, *) '␣'
            WRITE(TempUnitNum, '(I2)') Z
            DO Y = UBOUND(Cells,2), LBOUND(Cells,2), -1
                WRITE(TempUnitNum, fmt_domain) (Cells(:, Y, Z)%Centroid%Y)
            END DO
        END DO
        WRITE(TempUnitNum, *) '␣'
        WRITE(TempUnitNum, *) 'Cell␣Centroid␣Z'
        DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
            WRITE(TempUnitNum, *) '␣'
            WRITE(TempUnitNum, '(I2)') Z
            DO Y = UBOUND(Cells,2), LBOUND(Cells,2), -1
                WRITE(TempUnitNum, fmt_domain) (Cells(:, Y, Z)%Centroid%Z)
            END DO
        END DO
        WRITE(TempUnitNum, *) '␣'
        WRITE(TempUnitNum, *) 'Cell␣Depth'
        DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
            WRITE(TempUnitNum, *) '␣'
            WRITE(TempUnitNum, '(I2)') Z
            DO Y = UBOUND(Cells,2), LBOUND(Cells,2), -1
                WRITE(TempUnitNum, fmt_domain) (Extents%Ymax - Cells(:, Y, Z)%Centroid%Y)
            END DO
            END DO


            CLOSE(TempUnitNum)

        END IF

        !report the temperature field on an incremental time step
        IF (WriteTempProfiles) THEN
            IF (INT(REAL(TimeStepIndex)/REAL(FieldReportIncrement)) == REAL(TimeStepIndex)/REAL(
                 FieldReportIncrement)) THEN
                WRITE(TimeStepFileName, '(A23,␣I6,␣A7)') './GNUPlot/AfterTimeStep', TimeStepIndex, 'gnu.csv'
                OPEN(UNIT=TempUnitNum, STATUS='REPLACE', FILE=TimeStepFileName)
                !CALL FlushTemperatureField(TempUnitNum, fmt_domain)
                CALL FlushGNUPlotTemperatureField(TempUnitNum, INT(UBOUND(Cells,3)/2.0))
                CLOSE(TempUnitNum)
            END IF
        END IF

        !report the temperature field on the last time step
        IF (WriteTempProfiles) THEN
            IF (TimeStepIndex == NumTimeSteps) THEN
                OPEN(UNIT=TempUnitNum, STATUS='REPLACE', FILE='AfterLastTimeStep.csv')
                CALL FlushTemperatureField(TempUnitNum, fmt_domain)
                CLOSE(TempUnitNum)
            END IF
        END IF

        !write out the pipe circuit distribution each time step -- if there is a pipe circuit
        IF (Has%PipeCircuit) THEN

            !Do some one-time writing first
            IF (TimeStepIndex==1) THEN
                !Prepare format and write header to circuit profiles output
                WRITE(c_size, '(I4)') SIZE(PipeCircuit%ListOfCircuitPoints)
                WRITE(fmt_circuit, *) "("//TRIM(c_size)//"(F7.3,"",""))"
                WRITE(fmt_circuit2, *) "("//TRIM(c_size)//"(I7,"",""))"
                WRITE(CircuitProfilesUnitNum, fmt_circuit2) (i, i = LBOUND(PipeCircuit%ListOfCircuitPoints,1),
                     UBOUND(PipeCircuit%ListOfCircuitPoints,1))
                !Write header to circuit general output
                WRITE(CircuitUnitNum, '(A60)') 'Circuit␣Flow␣Rate,␣Circuit␣Cp,␣Circuit␣EFT,␣Circuit␣ExFT'
            END IF

            !Output the profile itself
            WRITE(CircuitProfilesUnitNum, fmt_circuit) ( Cells( PipeCircuit%ListOfCircuitPoints(i)%X, PipeCircuit
                 %ListOfCircuitPoints(i)%Y, PipeCircuit%ListOfCircuitPoints(i)%Z )%PipeCellData%Fluid%MyBase%
                 Temperature, i=LBOUND(PipeCircuit%ListOfCircuitPoints,1), UBOUND(PipeCircuit%ListOfCircuitPoints
                 ,1))

        END IF

        !now process the main output file
        WRITE(SimConditionsUnitNum, '(I6,␣",",␣I4,␣",",␣F4.1,␣",",␣F4.1,␣",",␣F4.1,␣",",␣F7.2)') &
                TimeStepIndex, NumIterationsUsed, CurAirTemp, CurRelativeHumidity, CurWindSpeed, CurIncidentSolar

        !if we have a pipe circuit, also report those characteristics
        IF (Has%PipeCircuit) THEN
          WRITE(CircuitUnitNum, '(F6.3,␣",",␣F6.1,␣",",␣F8.3,␣",",␣F8.3)') CurCircuitFlowRate,
               CurFluidSpecificHeat, PipeCircuit%CircuitInletCell%PipeCellData%Fluid%MyBase%Temperature,
               PipeCircuit%CircuitOutletCell%PipeCellData%Fluid%MyBase%Temperature
        END IF

        !report to the console periodically
        IF ((REAL(TimeStepIndex)/OutputInterval) == INT(TimeStepIndex/OutputInterval)) THEN
            WRITE(*,'(A20,␣I6,␣"/",␣I6)') "Completed␣TimeStep␣#", TimeStepIndex, NumTimeSteps
        END IF

    END DO

    !close output files
    IF (DoingUGTValidation) CLOSE(GroundTemperaturesUnitNum)
    IF (Has%PipeCircuit) CLOSE(CircuitProfilesUnitNum)
    CLOSE(SimConditionsUnitNum)

    !final processing and reporting
    WRITE(*, *) 'Simulation␣Completed'
    CALL DATE_AND_TIME(VALUES=DateTimeVals)
    WRITE(*, *) 'Ending␣Date␣and␣Time:'
    WRITE(*, '(I4,␣"-",␣I2,␣"-",␣I2,␣"␣␣--␣␣",␣I2,␣":",␣I2,␣":",␣I2)') &
        DateTimeVals(1), DateTimeVals(2), DateTimeVals(3), DateTimeVals(5), DateTimeVals(6), DateTimeVals(7)

    CONTINUE

END PROGRAM
```

Listing B.2: Standalone Ground Heat Exchanger Model Source: Data Structures

```
MODULE PiechowskiData

    IMPLICIT NONE
    PUBLIC

    !Any parameters which need to be commented once inside E+

    INTEGER, PARAMETER :: r64=KIND(1.0D0)
    REAL(r64), PARAMETER :: Pi = 3.1415926535d0

    character(len=*), parameter :: stopFile = 'stop.stop'

    !Define all Enumerations up here

    INTEGER, PARAMETER :: PartitionType_BasementWall = -1
    INTEGER, PARAMETER :: PartitionType_BasementFloor = -2
    INTEGER, PARAMETER :: PartitionType_Pipe = -3

    INTEGER, PARAMETER :: RegionType_Pipe = -1
    INTEGER, PARAMETER :: RegionType_BasementWall = -2
    INTEGER, PARAMETER :: RegionType_BasementFloor = -3
    INTEGER, PARAMETER :: RegionType_XDirection = -4
    INTEGER, PARAMETER :: RegionType_YDirection = -5
    INTEGER, PARAMETER :: RegionType_ZDirection = -6

    INTEGER, PARAMETER :: MeshDistribution_Uniform = -1
    INTEGER, PARAMETER :: MeshDistribution_SymmetricGeometric = -2

    INTEGER, PARAMETER :: SegmentFlow_IncreasingZ = -1
    INTEGER, PARAMETER :: SegmentFlow_DecreasingZ = -2

    INTEGER, PARAMETER :: Direction_PositiveY = -1
    INTEGER, PARAMETER :: Direction_NegativeY = -2
    INTEGER, PARAMETER :: Direction_PositiveX = -3
    INTEGER, PARAMETER :: Direction_NegativeX = -4
    INTEGER, PARAMETER :: Direction_PositiveZ = -5
    INTEGER, PARAMETER :: Direction_NegativeZ = -6

    INTEGER, PARAMETER :: CellType_Unknown = -1
    INTEGER, PARAMETER :: CellType_Pipe = -2
    INTEGER, PARAMETER :: CellType_GeneralField = -3
    INTEGER, PARAMETER :: CellType_GroundSurface = -4
    INTEGER, PARAMETER :: CellType_FarfieldBoundary = -5
    INTEGER, PARAMETER :: CellType_AdiabaticWall = -6
    INTEGER, PARAMETER :: CellType_BasementWall = -7
    INTEGER, PARAMETER :: CellType_BasementFloor = -8
    INTEGER, PARAMETER :: CellType_BasementCorner = -9
    INTEGER, PARAMETER :: CellType_BasementCutaway = -10

    INTEGER, PARAMETER :: BoundaryType_Adiabatic = -1
    INTEGER, PARAMETER :: BoundaryType_Farfield = -2

    INTEGER, PARAMETER :: FarfieldModel_Constant = -1
    INTEGER, PARAMETER :: FarfieldModel_ConstantLinear = -2
    INTEGER, PARAMETER :: FarfieldModel_KusudaAchenbach = -3


    !Other constants for convenience
    REAL(r64), PARAMETER :: SecondsInHour = 3600.0d0
    REAL(r64), PARAMETER :: MinutesInHour = 60.0d0
    REAL(r64), PARAMETER :: SecondsInMinute = 60.0d0
    REAL(r64), PARAMETER :: HoursInDay = 24.0d0
    REAL(r64), PARAMETER :: DaysInYear = 365.0d0
    REAL(r64), PARAMETER :: SecondsInDay = SecondsInHour * HoursInDay
    REAL(r64), PARAMETER :: SecondsInYear = SecondsInDay * DaysInYear

    TYPE BaseThermalPropertySet
      REAL(r64) :: Conductivity = 0.0d0  !W/mK
      REAL(r64) :: Density = 0.0d0       !kg/m3
      REAL(r64) :: SpecificHeat = 0.0d0  !J/kgK
    END TYPE

    TYPE ExtendedFluidProperties ! : Inherits BaseThermalPropertySet
      TYPE(BaseThermalPropertySet) :: MyBase
      REAL(r64) :: Viscosity  !kg/m-s
      REAL(r64) :: Prandtl    !-
    END TYPE

    TYPE ExtendedConstructionProperties ! : Inherits BaseThermalPropertySet
      TYPE(BaseThermalPropertySet) :: MyBase
      REAL(r64) :: Thickness !m
    END TYPE

    TYPE BaseCell
      REAL(r64) :: Temperature = 0.0d0 !C
      REAL(r64) :: Temperature_PrevIteration = 0.0d0 !C
      REAL(r64) :: Temperature_PrevTimeStep = 0.0d0 !C
      REAL(r64) :: Beta = 0.0d0 !K/W
      TYPE(BaseThermalPropertySet) :: Properties
```

```
END TYPE

TYPE RadialCellInformation ! : Inherits BaseCell
    TYPE(BaseCell) :: MyBase
    REAL(r64) :: RadialCentroid
    REAL(r64) :: InnerRadius
    REAL(r64) :: OuterRadius
END TYPE

TYPE FluidCellInformation ! : Inherits BaseCell
    TYPE(BaseCell) :: MyBase
    REAL(r64) :: PipeInnerRadius
    REAL(r64) :: Volume
    TYPE(ExtendedFluidProperties) :: Properties
END TYPE

TYPE CartesianPipeCellInformation
    TYPE(RadialCellInformation), ALLOCATABLE, DIMENSION(:) :: Soil
    TYPE(RadialCellInformation) :: Insulation
    TYPE(RadialCellInformation) :: Pipe
    TYPE(FluidCellInformation) :: Fluid
    REAL(r64) :: RadialSliceWidth
    REAL(r64) :: InterfaceVolume
END TYPE

TYPE Point
    INTEGER :: X
    INTEGER :: Y
END TYPE

TYPE PointF
    REAL(r64) :: X
    REAL(r64) :: Y
END TYPE

TYPE Point3DInteger
    INTEGER :: X
    INTEGER :: Y
    INTEGER :: Z
END TYPE

TYPE Point3DReal
    REAL(r64) :: X
    REAL(r64) :: Y
    REAL(r64) :: Z
END TYPE

TYPE DomainRectangle
    INTEGER :: XMin
    INTEGER :: XMax
    INTEGER :: Ymin
    INTEGER :: YMax
END TYPE

TYPE MeshPartition
    REAL(r64) :: rDimension
    INTEGER :: PartitionType !From Enum: ParitionType
    REAL(r64) :: TotalWidth
END TYPE

TYPE GridRegion
    REAL(r64) :: Min
    REAL(r64) :: Max
    INTEGER :: RegionType !From Enum: RegionType
    REAL(r64), ALLOCATABLE, DIMENSION(:) :: CellWidths
END TYPE

TYPE TempGridRegionData
    REAL(r64) :: Min
    REAL(r64) :: Max
    INTEGER   :: RegionType !From Enum: RegionType
END TYPE

TYPE RectangleF
    REAL(r64) :: X_min
    REAL(r64) :: Y_min
    REAL(r64) :: Width
    REAL(r64) :: Height
END TYPE

TYPE NeighborInformation
    REAL(r64) :: ThisCentroidToNeighborCentroid
    REAL(r64) :: ThisCentroidToNeighborWall
    REAL(r64) :: ThisWallToNeighborCentroid
    REAL(r64) :: ConductionResistance
    TYPE(Point3DInteger) :: NeighborCellIndeces
END TYPE

TYPE RadialSizing
    REAL(r64) :: InnerDia
    REAL(r64) :: OuterDia
END TYPE
```

```
CONTAINS

! Constructors for generic classes

SUBROUTINE CartesianPipeCellInformation_ctor(c, GridCellWidth, PipeSizes, NumRadialNodes, &
                                             CellDepth, InsulationThickness, RadialGridExtent,
                                                  SimHasInsulation)

    TYPE(CartesianPipeCellInformation), INTENT(IN OUT) :: c
    REAL(r64) :: GridCellWidth
    TYPE(RadialSizing) :: PipeSizes
    INTEGER :: NumRadialNodes
    REAL(r64) :: CellDepth
    REAL(r64) :: InsulationThickness
    REAL(r64) :: RadialGridExtent
    LOGICAL, INTENT(IN) :: SimHasInsulation

    REAL(r64) :: InsulationInnerRadius
    REAL(r64) :: InsulationOuterRadius
    REAL(r64) :: InsulationCentroid
    REAL(r64) :: PipeOuterRadius
    REAL(r64) :: PipeInnerRadius
    REAL(r64) :: MinimumSoilRadius
    REAL(r64) :: ThisSliceInnerRadius
    REAL(r64) :: Rval
    INTEGER :: RadialCellCtr

     !'calculate pipe radius
            PipeOuterRadius = PipeSizes%OuterDia / 2
            PipeInnerRadius = PipeSizes%InnerDia / 2

            !'--we will work from inside out, calculating dimensions and instantiating variables--
            !'first instantiate the water cell
            CALL FluidCellInformation_ctor(c%Fluid, PipeInnerRadius, CellDepth)

            !'then the pipe cell
    CALL RadialCellInformation_ctor(c%Pipe, (PipeOuterRadius + PipeInnerRadius) / 2.0d0, PipeInnerRadius,
         PipeOuterRadius)

            !'then the insulation if we have it
    IF (InsulationThickness > 0) THEN
        InsulationInnerRadius = PipeOuterRadius
        InsulationOuterRadius = InsulationInnerRadius + InsulationThickness
        InsulationCentroid = (InsulationInnerRadius + InsulationOuterRadius) / 2.0d0
        CALL RadialCellInformation_ctor(c%Insulation, InsulationCentroid, InsulationInnerRadius,
             InsulationOuterRadius)
    END IF

            !'determine where to start applying the radial soil cells based on whether we have insulation or
                 not
            IF (.NOT. SimHasInsulation) THEN
        MinimumSoilRadius = PipeOuterRadius
            ELSE
                    MinimumSoilRadius = c%Insulation%OuterRadius
            END IF

            !'the radial cells are distributed evenly throughout this region
            c%RadialSliceWidth = RadialGridExtent / NumRadialNodes

            !allocate the array of radial soil nodes
            ALLOCATE(c%Soil(0:NumRadialNodes - 1))

            !first set Rval to the minimum soil radius plus half a slice thickness for the innermost radial
                 node
            Rval = MinimumSoilRadius + (c%RadialSliceWidth / 2.0d0)
            ThisSliceInnerRadius = MinimumSoilRadius
    CALL RadialCellInformation_ctor(c%Soil(0), Rval, ThisSliceInnerRadius, ThisSliceInnerRadius + c%
         RadialSliceWidth)

            !'then loop through the rest and assign them, each radius is simply one more slice thickness
            DO RadialCellCtr = 1, UBOUND(c%Soil,1)
                    Rval = Rval + c%RadialSliceWidth
                    ThisSliceInnerRadius = ThisSliceInnerRadius + c%RadialSliceWidth
                    CALL RadialCellInformation_ctor(c%Soil(RadialCellCtr), Rval, ThisSliceInnerRadius,
                         ThisSliceInnerRadius + c%RadialSliceWidth)
            END DO

            !'also assign the interface cell surrounding the radial system
            c%InterfaceVolume = (1.0d0 - (3.1415926535d0 / 4.0d0)) * (GridCellWidth ** 2) * CellDepth

    RETURN

END SUBROUTINE

SUBROUTINE RadialCellInformation_ctor(c, m_RadialCentroid, m_MinRadius, m_MaxRadius)

    TYPE(RadialCellInformation), INTENT(IN OUT) :: c

    REAL(r64) :: m_RadialCentroid
            REAL(r64) :: m_MinRadius
            REAL(r64) :: m_MaxRadius

    c%RadialCentroid = m_RadialCentroid
```

```
                c%InnerRadius = m_MinRadius
                c%OuterRadius = m_MaxRadius

        RETURN

    END SUBROUTINE

    SUBROUTINE FluidCellInformation_ctor(c, m_PipeInnerRadius, m_CellDepth)

        TYPE(FluidCellInformation), INTENT(IN OUT) :: c
        REAL(r64) :: m_PipeInnerRadius
        REAL(r64) :: m_CellDepth

        c%PipeInnerRadius = m_PipeInnerRadius
        c%Volume = 3.1415926535d0 * (m_PipeInnerRadius ** 2) * m_CellDepth

        RETURN

    END SUBROUTINE

END MODULE

MODULE mCartesianCell ! : Inherits BaseCell

    USE PiechowskiData
    IMPLICIT NONE
    PUBLIC

    TYPE DirectionNeighbor_Dictionary
        INTEGER :: Direction !From Enum: Direction
        TYPE(NeighborInformation) :: Value
    END TYPE

    TYPE CartesianCell
        TYPE(BaseCell) :: MyBase
        INTEGER :: X_index
        INTEGER :: Y_index
        INTEGER :: Z_index
        REAL(r64) :: X_min
        REAL(r64) :: X_max
        REAL(r64) :: Y_min
        REAL(r64) :: Y_max
        REAL(r64) :: Z_min
        REAL(r64) :: Z_max
        TYPE(Point3DReal) :: Centroid
        INTEGER :: CellType !From Enum: CellType
        INTEGER :: PipeIndex
        TYPE(DirectionNeighbor_Dictionary), ALLOCATABLE, DIMENSION(:) :: NeighborInformation
        TYPE(CartesianPipeCellInformation) :: PipeCellData
    END TYPE

    CONTAINS

    REAL(r64) FUNCTION Width(c) RESULT (RetVal)
        TYPE(CartesianCell), INTENT(IN) :: c
        RetVal = c%X_max - c%X_min
        RETURN
    END FUNCTION

    REAL(r64) FUNCTION Height(c) RESULT (RetVal)
        TYPE(CartesianCell), INTENT(IN) :: c
        RetVal = c%Y_max - c%Y_min
        RETURN
    END FUNCTION

    REAL(r64) FUNCTION Depth(c) RESULT (RetVal)
        TYPE(CartesianCell), INTENT(IN) :: c
        RetVal = c%Z_max - c%Z_min
        RETURN
    END FUNCTION

    REAL(r64) FUNCTION XNormalArea(c) RESULT (RetVal)
        TYPE(CartesianCell), INTENT(IN) :: c
        RetVal = Depth(c) * Height(c)
        RETURN
    END FUNCTION

    REAL(r64) FUNCTION YNormalArea(c) RESULT (RetVal)
        TYPE(CartesianCell), INTENT(IN) :: c
        RetVal = Depth(c) * Width(c)
        RETURN
    END FUNCTION

    REAL(r64) FUNCTION ZNormalArea(c) RESULT (RetVal)
        TYPE(CartesianCell), INTENT(IN) :: c
        RetVal = Width(c) * Height(c)
        RETURN
    END FUNCTION

    REAL(r64) FUNCTION Volume(c) RESULT (RetVal)
        TYPE(CartesianCell), INTENT(IN) :: c
        RetVal = Width(c) * Depth(c) * Height(c)
        RETURN
    END FUNCTION

    TYPE(RectangleF) FUNCTION XYRectangle(c) RESULT (RetVal)
        TYPE(CartesianCell), INTENT(IN) :: c
        RetVal = RectangleF(c%X_min, c%Y_min, Width(c), Height(c))
        RETURN
    END FUNCTION

    TYPE(RectangleF) FUNCTION XZRectangle(c) RESULT (RetVal)
        TYPE(CartesianCell), INTENT(IN) :: c
        RetVal = RectangleF(c%X_min, c%Z_min, Width(c), Depth(c))
        RETURN
    END FUNCTION

    TYPE(RectangleF) FUNCTION YZRectangle(c) RESULT (RetVal)
        TYPE(CartesianCell), INTENT(IN) :: c
        RetVal = RectangleF(c%Y_min, c%Z_min, Height(c), Depth(c))
        RETURN
    END FUNCTION

    REAL(r64) FUNCTION NormalArea(c, Direction) RESULT (RetVal)
        TYPE(CartesianCell), INTENT(IN) :: c
        INTEGER, INTENT(IN) :: Direction !From Enum: Direction
        SELECT CASE (Direction)
        CASE (Direction_PositiveY, Direction_NegativeY)
            RetVal = YNormalArea(c)
        CASE (Direction_PositiveX, Direction_NegativeX)
            RetVal = XNormalArea(c)
        CASE (Direction_PositiveZ, Direction_NegativeZ)
            RetVal = ZNormalArea(c)
        END SELECT
        RETURN
    END FUNCTION

    TYPE(NeighborInformation) FUNCTION NeighborInformationArray_Value(dict, direction) RESULT(RetVal)

        TYPE(DirectionNeighbor_Dictionary), ALLOCATABLE, DIMENSION(:) :: dict
        INTEGER :: Direction !From Enum: Direction

        INTEGER :: Index

        DO Index = LBOUND(dict,1), UBOUND(dict,1)
            IF (dict(Index)%Direction == direction) THEN
                RetVal = dict(Index)%Value
                EXIT
            END IF
        END DO

        RETURN

    END FUNCTION

END MODULE


MODULE Extensions

    USE PiechowskiData
    IMPLICIT NONE
    PUBLIC

    INTERFACE IsInRange
      MODULE PROCEDURE Integer_IsInRange
      MODULE PROCEDURE Real_IsInRange
    END INTERFACE

    CONTAINS

    LOGICAL FUNCTION Integer_IsInRange(i, lower, upper) RESULT(RetVal)

        INTEGER, INTENT(IN) :: i, lower, upper

        IF ((i >= lower) .AND. (i <= upper)) THEN
            RetVal = .TRUE.
        ELSE
            RetVal = .FALSE.
        END IF

        RETURN

    END FUNCTION

    LOGICAL FUNCTION Real_IsInRange(r, lower, upper) RESULT(RetVal)

        REAL(r64), INTENT(IN) :: r, lower, upper

        IF ((r >= lower) .AND. (r <= upper)) THEN
            RetVal = .TRUE.
        ELSE
            RetVal = .FALSE.
        END IF

        RETURN
```

```
END FUNCTION

LOGICAL FUNCTION CellType_IsFieldCell(CellType) RESULT(RetVal)

    INTEGER, INTENT(IN) :: CellType !From Enum: CellType

    SELECT CASE (CellType)
    CASE (CellType_GeneralField, CellType_BasementCorner, &
          CellType_BasementWall, CellType_BasementFloor)
        RetVal = .TRUE.
    CASE DEFAULT
        RetVal = .FALSE.
    END SELECT

    RETURN

END FUNCTION

REAL(r64) FUNCTION Real_ConstrainTo(r, MinVal, MaxVal) RESULT(RetVal)

    REAL(r64), INTENT(IN) :: r, MinVal, MaxVal

    RetVal = MIN(r, MaxVal)
    RetVal = MAX(r, MinVal)

    RETURN

END FUNCTION

LOGICAL FUNCTION MeshPartitionArray_Contains(meshes, value) RESULT(RetVal)

    TYPE(MeshPartition), ALLOCATABLE, DIMENSION(:), INTENT(IN) :: meshes
    REAL(r64), INTENT(IN) :: value

    INTEGER :: meshnum

    RetVal = .FALSE.

    DO meshnum = LBOUND(meshes, 1), UBOUnD(meshes, 1)
        IF (meshes(meshnum)%rDimension == value) THEN
            RetVal = .TRUE.
            EXIT
        END IF
    END DO

    RETURN

END FUNCTION

REAL(r64) FUNCTION RadialCellInfo_XY_CrossSectArea(r) RESULT (RetVal)

    TYPE(RadialCellInformation), INTENT(IN) :: r

    RetVal = 3.14159d0 * ((r%OuterRadius**2) - (r%InnerRadius**2))

    RETURN

END FUNCTION

LOGICAL FUNCTION DomainRectangle_Contains(Rect, p) RESULT(RetVal)

    TYPE(DomainRectangle), INTENT(IN) :: Rect
    TYPE(Point), INTENT(IN) :: p

    IF (IsInRange(p%X, Rect%XMin, Rect%XMax) .AND. IsInRange(p%Y, Rect%YMin, Rect%YMax)) THEN
        RetVal = .TRUE.
    ELSE
        RetVal = .FALSE.
    END IF

    RETURN

END FUNCTION

SUBROUTINE MeshPartition_SelectionSort(X)

    TYPE(MeshPartition), ALLOCATABLE, DIMENSION(:), INTENT(IN OUT) :: X
    TYPE(MeshPartition) :: TEMP
    INTEGER :: I, ISWAP(1), ITEMP, ISWAP1

    DO I = LBOUND(X, 1), UBOUND(X, 1)-1
        ISWAP=MINLOC(X(I:)%rDimension)
        ISWAP1=ISWAP(1)+I-1
        IF(ISWAP1.NE.I) THEN
            TEMP=X(I)
            X(I)=X(ISWAP1)
            X(ISWAP1)=TEMP
        ENDIF
    END DO

    RETURN
    END SUBROUTINE

    INTEGER FUNCTION MeshPartition_CompareByDimension(x, y) RESULT(RetVal)

        TYPE(MeshPartition), INTENT(IN) :: x
        TYPE(MeshPartition), INTENT(IN) :: y

        IF (x%rDimension < y%rDimension) THEN
          RetVal = -1
        ELSEIF (x%rDimension > y%rDimension) THEN
          RetVal = 1
        ELSE
          RetVal = 0
        ENDIF

        RETURN

    END FUNCTION

    REAL(r64) FUNCTION BaseThermalPropertySet_Diffusivity(p) RESULT(RetVal)

        TYPE(BaseThermalPropertySet), INTENT(IN) :: p

        RetVal = p%Conductivity / (p%Density * p%SpecificHeat)

        RETURN

    END FUNCTION

    LOGICAL FUNCTION RectangleF_Contains(rect, p) RESULT(RetVal)

        TYPE(RectangleF), INTENT(IN) :: rect
        TYPE(PointF), INTENT(IN) :: p

        RetVal = ((((Rect%X_min <= p%X) .AND. (p%X < (Rect%X_min + rect%Width))) .AND. (rect%Y_min <= p%Y)) .AND.
              (p%Y < (rect%Y_min + rect%Height)))

        RETURN

    END FUNCTION

END MODULE

MODULE Sim

    USE PiechowskiData
    USE mCartesianCell
    IMPLICIT NONE
    PUBLIC

    ! Structure based (loosely) on input data

    TYPE MeshExtents
        REAL(r64) :: Xmax
        REAL(r64) :: Ymax
        REAL(r64) :: Zmax
    END TYPE

    TYPE DistributionStructure
        INTEGER :: MeshDistribution !From Enum: MeshDistribution
        INTEGER :: RegionMeshCount
        REAL(r64) :: GeometricSeriesCoefficient
        INTEGER :: BoundaryType !From Enum: BoundaryType
    END TYPE
    TYPE RadialMeshStructure
        INTEGER :: NumRadialCells
        REAL(r64) :: RadialMeshThickness
    END TYPE
    TYPE MeshProperties
        TYPE(DistributionStructure) :: X
        TYPE(DistributionStructure) :: Y
        TYPE(DistributionStructure) :: Z
        TYPE(RadialMeshStructure) :: Radial
    END TYPE

    TYPE PipeSegmentInfo
        TYPE(PointF) :: PipeLocation
        INTEGER      :: FlowDirection !From Enum: SegmentFlow
        TYPE(Point)  :: PipeCellCoordinates
    END TYPE
    TYPE PipeCircuitInfo
        TYPE(CartesianCell), POINTER :: CircuitInletCell
        TYPE(CartesianCell), POINTER :: CircuitOutletCell
        TYPE(RadialSizing) :: PipeSize
        TYPE(RadialSizing) :: InsulationSize
        TYPE(PipeSegmentInfo), ALLOCATABLE, DIMENSION(:) :: PipeSegments
        REAL(r64) :: CurCircuitFlowRate
        REAL(r64) :: CurCircuitConvectionCoefficient
        TYPE(ExtendedFluidProperties) :: CurFluidPropertySet
        TYPE(Point3DInteger), ALLOCATABLE, DIMENSION(:) :: ListOfCircuitPoints
    END TYPE

    TYPE TransientInfo
        REAL(r64) :: Convergence_CurrentToPrevIteration
        INTEGER :: MaxIterationsPerTS
    END TYPE
    TYPE SimulationControl
        REAL(r64) :: DomainInitialTemperature
        REAL(r64) :: MinimumTemperatureLimit = -1000
        REAL(r64) :: MaximumTemperatureLimit = 1000
        TYPE(TransientInfo) :: Cartesian
        TYPE(TransientInfo) :: Radial
    END TYPE

    TYPE Farfield_Constant
        REAL(r64) :: Temperature !C
    END TYPE
    TYPE Farfield_ConstantLinear
        REAL(r64) :: SurfaceTemperature !C
        REAL(r64) :: Slope !C/[scaled depth]
    END TYPE
    TYPE Farfield_KusudaAchenbach
        REAL(r64) :: AverageGroundTemperature !C
        REAL(r64) :: AverageGroundTemperatureAmplitude !C
        REAL(r64) :: PhaseShiftOfMinGroundTempDays !days
        REAL(r64) :: PhaseShiftOfMinGroundTemp !seconds
    END TYPE
    TYPE FarfieldInfo
        INTEGER :: Model !From Enum: FarfieldModel
        TYPE(Farfield_Constant) :: Constant
        TYPE(Farfield_ConstantLinear) :: ConstantLinear
        TYPE(Farfield_KusudaAchenbach) :: KusudaAchenbach
    END TYPE

    TYPE BasementZoneInfo
        REAL(r64) :: Depth !m
        REAL(r64) :: Width
        LOGICAL :: ShiftPipesByWidth
        INTEGER :: UnderBasementBoundaryType !From Enum: BoundaryType
        TYPE(ExtendedConstructionProperties) :: BasementFloor
        TYPE(ExtendedConstructionProperties) :: BasementWall
        REAL(r64) :: CurBasementTemperature
        REAL(r64) :: CurBasementWallConvectionCoeff
        REAL(r64) :: CurBasementFloorConvectionCoeff
    END TYPE

    ! "Input" data structure variables
    TYPE(MeshExtents) :: Extents
    TYPE(MeshProperties) :: Mesh
    TYPE(PipeCircuitInfo), SAVE :: PipeCircuit
    TYPE(BaseThermalPropertySet), SAVE :: GroundProperties
    TYPE(BaseThermalPropertySet), SAVE :: PipeProperties
    TYPE(BaseThermalPropertySet), SAVE :: InsulationProperties
    TYPE(SimulationControl), SAVE :: SimControls
    TYPE(FarfieldInfo) :: Farfield
    TYPE(BasementZoneInfo), SAVE :: BasementZone

    ! Internal structure

    TYPE HasStuff
        LOGICAL :: Basement
        LOGICAL :: PipeCircuit
        LOGICAL :: Insulation
    END TYPE
    TYPE(HasStuff) :: Has

    TYPE DirectionReal_Dictionary
        INTEGER :: Direction !From Enum: Direction
        REAL(r64) :: Value
    END TYPE
    TYPE ReportingInformation
        TYPE(DirectionReal_Dictionary), ALLOCATABLE, DIMENSION(:) :: SurfaceHeatTransfer
        REAL(r64) :: TotalBoundaryHeatTransfer
        REAL(r64) :: EnergyStoredInCells
        REAL(r64) :: AverageSurfaceTemperature
        REAL(r64) :: PipeCircuitHeatTransferMCpDT
        REAL(r64) :: PipeCircuitHeatTransferUADT
        REAL(r64) :: BasementWallHeatTransfer
        REAL(r64) :: BasementFloorHeatTransfer
        REAL(r64) :: AverageBasementFloorTemperature
        REAL(r64) :: AverageBasementWallTemperature
    END TYPE
    TYPE(ReportingInformation) :: Reporting

    TYPE(CartesianCell), ALLOCATABLE, DIMENSION(:,:,:), TARGET :: Cells

    TYPE MeshPartitions
        TYPE(MeshPartition), ALLOCATABLE, DIMENSION(:) :: X
        TYPE(MeshPartition), ALLOCATABLE, DIMENSION(:) :: Y
    END TYPE
    TYPE(MeshPartitions) :: Partitions

    INTEGER :: BasementWallXIndex = -1
    INTEGER :: BasementFloorYIndex = -1

    CONTAINS
```

```
!Extension methods for Sim classes

REAL(r64) FUNCTION RadialSizing_Thickness(r) RESULT (RetVal)
    TYPE(RadialSizing), INTENT(IN) :: r
    RetVal = (r%OuterDia - r%InnerDia) / 2.0d0
    RETURN
END FUNCTION

SUBROUTINE PipeSegmentInfo_InitPipeCells(s, x, y)
    TYPE(PipeSegmentInfo), INTENT(IN OUT) :: s
    INTEGER, INTENT(IN) :: x, y
    TYPE(Point) :: TempPt
    TempPt%X = x
    TempPt%Y = y
    s%PipeCellCoordinates = TempPt
    RETURN
END SUBROUTINE

SUBROUTINE PipeCircuitInfo_InitInOutCells(c, in, out)
    TYPE(PipeCircuitInfo), INTENT(IN OUT) :: c
    TYPE(CartesianCell), INTENT(IN), TARGET :: in, out
    c%CircuitInletCell => in
    c%CircuitOutletCell => out
    RETURN
END SUBROUTINE

! Convergence checking
LOGICAL FUNCTION IsConverged_CurrentToPrevIteration(MaxDivAmount) RESULT (RetVal)

    REAL(r64), INTENT(IN OUT) :: MaxDivAmount

    REAL(r64) :: LocalMax, ThisCellMax
    INTEGER :: X, Y, Z
    TYPE(CartesianCell) :: ThisCell
    !REAL(r64) :: MaxPipeCellDeviation
    !LOGICAL :: ThisPipeCellConverged

    LocalMax = 0.0d0
    DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
        DO Y = LBOUND(Cells,2), UBOUND(Cells,2)
            DO X = LBOUND(Cells,1), UBOUND(Cells,1)
                ThisCell = Cells(X, Y, Z)
                ThisCellMax = ABS(ThisCell%MyBase%Temperature - ThisCell%MyBase%Temperature_PrevIteration)
                LocalMax = MAX(LocalMax, ThisCellMax)
            END DO
        END DO
    END DO

    RetVal = (LocalMax < SimControls%Cartesian%Convergence_CurrentToPrevIteration)

    RETURN

END FUNCTION

LOGICAL FUNCTION IsConverged_PipeCurrentToPrevIteration(CellToCheck, MaxDivAmount) RESULT (RetVal)

    TYPE(CartesianCell), INTENT(IN) :: CellToCheck
    REAL(r64), INTENT(IN OUT) :: MaxDivAmount

    INTEGER :: RadialCtr
    REAL(r64) :: ThisCellMax
    TYPE(RadialCellInformation) :: radCell

    MaxDivAmount = 0.0d0
            DO RadialCtr = LBOUND(CellToCheck%PipeCellData%Soil,1), UBOUND(CellToCheck%PipeCellData%Soil,1)
                    radCell = CellToCheck%PipeCellData%Soil(RadialCtr)
        ThisCellMax = ABS(radCell%MyBase%Temperature - radCell%MyBase%Temperature_PrevIteration)
                    IF (ThisCellMax > MaxDivAmount) THEN
                            MaxDivAmount = ThisCellMax
        END IF
    END DO
            !'also do the pipe cell
            ThisCellMax = ABS(CellToCheck%PipeCellData%Pipe%MyBase%Temperature - CellToCheck%PipeCellData%
                Pipe%MyBase%Temperature_PrevIteration)
            IF (ThisCellMax > MaxDivAmount) THEN
                    MaxDivAmount = ThisCellMax
            END IF
            !'also do the water cell
            ThisCellMax = ABS(CellToCheck%PipeCellData%Fluid%MyBase%Temperature - CellToCheck%PipeCellData%
                Fluid%MyBase%Temperature_PrevIteration)
            IF (ThisCellMax > MaxDivAmount) THEN
                    MaxDivAmount = ThisCellMax
            END IF
            !'also do insulation if it exists
            IF (Has%Insulation) THEN
                    ThisCellMax = ABS(CellToCheck%PipeCellData%Insulation%MyBase%Temperature - CellToCheck%
                        PipeCellData%Insulation%MyBase%Temperature_PrevIteration)
                    IF (ThisCellMax > MaxDivAmount) THEN
                            MaxDivAmount = ThisCellMax
                    END IF
            END IF
            RetVal = (MaxDivAmount < SimControls%Radial%Convergence_CurrentToPrevIteration)

    RETURN

    END FUNCTION

!Write temperature field to file lun using format fmt
SUBROUTINE FlushTemperatureField(lun, fmt)

    INTEGER, INTENT(IN) :: lun !Unit number to write temperature field
    CHARACTER(*), INTENT(IN) :: fmt

    INTEGER :: Y
    INTEGER :: Z

    DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
        WRITE(lun, *) '␣'
        WRITE(lun, '(I2)') Z
        DO Y = UBOUND(Cells,2), LBOUND(Cells,2), -1
            WRITE(lun, fmt) (Cells(:, Y, Z)%MyBase%Temperature)
        END DO
    END DO

END SUBROUTINE

!Write temperature field to file lun using format fmt
SUBROUTINE FlushGNUPlotTemperatureField(lun, Z)

    INTEGER, INTENT(IN) :: lun !Unit number to write temperature field
    INTEGER, INTENT(IN) :: Z !index of z to report

    INTEGER :: X
    INTEGER :: Y

    DO Y = UBOUND(Cells,2), LBOUND(Cells,2), -1
        DO X = LBOUND(Cells,1), UBOUND(Cells,1)
            WRITE(lun, '(3(F7.3,␣",")))') Cells(X, Y, Z)%Centroid%X, Cells(X, Y, Z)%Centroid%Y, Cells(X, Y, Z)
                 %MyBase%Temperature
        END DO
        WRITE(lun, *) '␣'
    END DO

END SUBROUTINE

! Set cell array temperature values
SUBROUTINE SetAllCellTempsToValue(NewTemp)

    REAL(r64) :: NewTemp

    INTEGER :: X, Y, Z, RadCtr
    TYPE(CartesianCell) :: ThisCell

    DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
        DO Y = LBOUND(Cells,2), UBOUND(Cells,2)
            DO X = LBOUND(Cells,1), UBOUND(Cells,1)
                Cells(X, Y, Z)%MyBase%Temperature = NewTemp
                IF (Cells(X, Y, Z)%CellType == CellType_Pipe) THEN
                    DO RadCtr = LBOUND(Cells(X, Y, Z)%PipeCellData%Soil, 1), UBOUND(Cells(X, Y, Z)%
                         PipeCellData%Soil, 1)
                        Cells(X, Y, Z)%PipeCellData%Soil(RadCtr)%MyBase%Temperature = NewTemp
                    END DO
                    Cells(X, Y, Z)%PipeCellData%Fluid%MyBase%Temperature = NewTemp
                    Cells(X, Y, Z)%PipeCellData%Pipe%MyBase%Temperature = NewTemp
                    IF (Has%Insulation) THEN
                        Cells(X, Y, Z)%PipeCellData%Insulation%MyBase%Temperature = NewTemp
                    END IF
                END IF
            END DO
        END DO
    END DO

    RETURN

END SUBROUTINE

SUBROUTINE SetAllCellTempPrevItersToValue(NewTemp)

    REAL(r64) :: NewTemp

    INTEGER :: X, Y, Z, RadCtr
    TYPE(CartesianCell) :: ThisCell

    DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
        DO Y = LBOUND(Cells,2), UBOUND(Cells,2)
            DO X = LBOUND(Cells,1), UBOUND(Cells,1)
                Cells(X, Y, Z)%MyBase%Temperature_PrevIteration = NewTemp
                IF (Cells(X, Y, Z)%CellType == CellType_Pipe) THEN
                    DO RadCtr = LBOUND(Cells(X, Y, Z)%PipeCellData%Soil, 1), UBOUND(Cells(X, Y, Z)%
                         PipeCellData%Soil, 1)
                        Cells(X, Y, Z)%PipeCellData%Soil(RadCtr)%MyBase%Temperature_PrevIteration = NewTemp
                    END DO
                    Cells(X, Y, Z)%PipeCellData%Fluid%MyBase%Temperature_PrevIteration = NewTemp
                    Cells(X, Y, Z)%PipeCellData%Pipe%MyBase%Temperature_PrevIteration = NewTemp
                    IF (Has%Insulation) THEN
                        Cells(X, Y, Z)%PipeCellData%Insulation%MyBase%Temperature_PrevIteration = NewTemp
                    END IF
                END IF
            END DO
        END DO
    END DO

    RETURN

END SUBROUTINE

SUBROUTINE SetAllCellTempPrevTimesToValue(NewTemp)

    REAL(r64) :: NewTemp

    INTEGER :: X, Y, Z, RadCtr
    TYPE(CartesianCell) :: ThisCell

    DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
        DO Y = LBOUND(Cells,2), UBOUND(Cells,2)
            DO X = LBOUND(Cells,1), UBOUND(Cells,1)
                Cells(X, Y, Z)%MyBase%Temperature_PrevTimeStep = NewTemp
                IF (Cells(X, Y, Z)%CellType == CellType_Pipe) THEN
                    DO RadCtr = LBOUND(Cells(X, Y, Z)%PipeCellData%Soil, 1), UBOUND(Cells(X, Y, Z)%
                         PipeCellData%Soil, 1)
                        Cells(X, Y, Z)%PipeCellData%Soil(RadCtr)%MyBase%Temperature_PrevTimeStep = NewTemp
                    END DO
                    Cells(X, Y, Z)%PipeCellData%Fluid%MyBase%Temperature_PrevTimeStep = NewTemp
                    Cells(X, Y, Z)%PipeCellData%Pipe%MyBase%Temperature_PrevTimeStep = NewTemp
                    IF (Has%Insulation) THEN
                        Cells(X, Y, Z)%PipeCellData%Insulation%MyBase%Temperature_PrevTimeStep = NewTemp
                    END IF
                END IF
            END DO
        END DO
    END DO

    RETURN

END SUBROUTINE

SUBROUTINE SetAllCellTempTypesToValue(NewTemp)

    REAL(r64) :: NewTemp

            CALL SetAllCellTempPrevItersToValue(NewTemp)
            CALL SetAllCellTempPrevTimesToValue(NewTemp)
            CALL SetAllCellTempsToValue(NewTemp)

    RETURN

    END SUBROUTINE

SUBROUTINE ResetAllCellTempTypesToDefault()
            CALL SetAllCellTempTypesToValue(SimControls%DomainInitialTemperature)
    END SUBROUTINE

SUBROUTINE ShiftTemperaturesForNewTimeStep()

    INTEGER :: X, Y, Z, RadCtr
    TYPE(CartesianCell) :: ThisCell

    DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
        DO Y = LBOUND(Cells,2), UBOUND(Cells,2)
            DO X = LBOUND(Cells,1), UBOUND(Cells,1)
                Cells(X, Y, Z)%MyBase%Temperature_PrevTimeStep = Cells(X, Y, Z)%MyBase%Temperature
                IF (Cells(X, Y, Z)%CellType == CellType_Pipe) THEN
                    DO RadCtr = LBOUND(Cells(X, Y, Z)%PipeCellData%Soil, 1), UBOUND(Cells(X, Y, Z)%
                         PipeCellData%Soil, 1)
                        Cells(X, Y, Z)%PipeCellData%Soil(RadCtr)%MyBase%Temperature_PrevTimeStep = Cells(X, Y
                             , Z)%PipeCellData%Soil(RadCtr)%MyBase%Temperature
                    END DO
                    Cells(X, Y, Z)%PipeCellData%Fluid%MyBase%Temperature_PrevTimeStep = Cells(X, Y, Z)%
                         PipeCellData%Fluid%MyBase%Temperature
                    Cells(X, Y, Z)%PipeCellData%Pipe%MyBase%Temperature_PrevTimeStep = Cells(X, Y, Z)%
                         PipeCellData%Pipe%MyBase%Temperature
                    IF (Has%Insulation) THEN
                        Cells(X, Y, Z)%PipeCellData%Insulation%MyBase%Temperature_PrevTimeStep = Cells(X, Y,
                             Z)%PipeCellData%Insulation%MyBase%Temperature
                    END IF
                END IF
            END DO
        END DO
    END DO

    RETURN

END SUBROUTINE

SUBROUTINE ShiftTemperaturesForNewIteration()

    INTEGER :: X, Y, Z, RadCtr
    TYPE(CartesianCell) :: ThisCell

    DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
        DO Y = LBOUND(Cells,2), UBOUND(Cells,2)
            DO X = LBOUND(Cells,1), UBOUND(Cells,1)
                Cells(X, Y, Z)%MyBase%Temperature_PrevIteration = Cells(X, Y, Z)%MyBase%Temperature
                IF (Cells(X, Y, Z)%CellType == CellType_Pipe) THEN
                    DO RadCtr = LBOUND(Cells(X, Y, Z)%PipeCellData%Soil, 1), UBOUND(Cells(X, Y, Z)%
                         PipeCellData%Soil, 1)
                        Cells(X, Y, Z)%PipeCellData%Soil(RadCtr)%MyBase%Temperature_PrevIteration = Cells(X,
                             Y, Z)%PipeCellData%Soil(RadCtr)%MyBase%Temperature
                    END DO
                    Cells(X, Y, Z)%PipeCellData%Fluid%MyBase%Temperature_PrevIteration = Cells(X, Y, Z)%
                         PipeCellData%Fluid%MyBase%Temperature
                    Cells(X, Y, Z)%PipeCellData%Pipe%MyBase%Temperature_PrevIteration = Cells(X, Y, Z)%
                         PipeCellData%Pipe%MyBase%Temperature
                    IF (Has%Insulation) THEN
                        Cells(X, Y, Z)%PipeCellData%Insulation%MyBase%Temperature_PrevIteration = Cells(X, Y,
                              Z)%PipeCellData%Insulation%MyBase%Temperature
                    END IF
                END IF
            END DO
        END DO
    END DO

    RETURN

END SUBROUTINE

SUBROUTINE ShiftPipeTemperaturesForNewIteration(ThisPipeCell)

    TYPE(CartesianCell), INTENT(IN OUT) :: ThisPipeCell

    INTEGER :: RadCtr

    IF (ThisPipeCell%CellType == CellType_Pipe) THEN !It better be!
        DO RadCtr = LBOUND(ThisPipeCell%PipeCellData%Soil, 1), UBOUND(ThisPipeCell%PipeCellData%Soil, 1)
            ThisPipeCell%PipeCellData%Soil(RadCtr)%MyBase%Temperature_PrevIteration = ThisPipeCell%
                 PipeCellData%Soil(RadCtr)%MyBase%Temperature
        END DO
        ThisPipeCell%PipeCellData%Fluid%MyBase%Temperature_PrevIteration = ThisPipeCell%PipeCellData%Fluid%
             MyBase%Temperature
        ThisPipeCell%PipeCellData%Pipe%MyBase%Temperature_PrevIteration = ThisPipeCell%PipeCellData%Pipe%
             MyBase%Temperature
        IF (Has%Insulation) THEN
            ThisPipeCell%PipeCellData%Insulation%MyBase%Temperature_PrevIteration = ThisPipeCell%PipeCellData
                 %Insulation%MyBase%Temperature
        END IF
    END IF

    RETURN

END SUBROUTINE

SUBROUTINE SetAllPipeTemperaturesToValue(ThisPipeCell, NewTemp, NewFluidTemp)

    TYPE(CartesianCell), INTENT(IN OUT) :: ThisPipeCell
    REAL(r64) :: NewTemp
    REAL(r64), OPTIONAL :: NewFluidTemp

    INTEGER :: RadCtr

    IF (ThisPipeCell%CellType == CellType_Pipe) THEN !It better be!
        DO RadCtr = LBOUND(ThisPipeCell%PipeCellData%Soil, 1), UBOUND(ThisPipeCell%PipeCellData%Soil, 1)
            ThisPipeCell%PipeCellData%Soil(RadCtr)%MyBase%Temperature = NewTemp
        END DO
        IF (PRESENT(NewFluidTemp)) THEN
            ThisPipeCell%PipeCellData%Fluid%MyBase%Temperature = NewFluidTemp
        ELSE
            ThisPipeCell%PipeCellData%Fluid%MyBase%Temperature = NewTemp
        END IF
        ThisPipeCell%PipeCellData%Pipe%MyBase%Temperature = NewTemp
        IF (Has%Insulation) THEN
            ThisPipeCell%PipeCellData%Insulation%MyBase%Temperature = NewTemp
        END IF
        ThisPipeCell%MyBase%Temperature = NewTemp
    END IF

    RETURN

END SUBROUTINE

LOGICAL FUNCTION CheckForOutOfRangeTemps() RESULT (RetVal)

    IF (ANY(Cells%MyBase%Temperature .GT. SimControls%MaximumTemperatureLimit) .OR. &
        ANY(Cells%MyBase%Temperature .LT. SimControls%MinimumTemperatureLimit)) THEN
        RetVal = .TRUE.
    ELSE
        RetVal = .FALSE.
    END IF

    RETURN
    END FUNCTION

    INTEGER FUNCTION GetAvailableNeighborCountForThisCell(c) RESULT (RetVal)

        TYPE(CartesianCell), INTENT(IN) :: c

        INTEGER :: x, y, z

        x = c%X_index
        y = c%Y_index
        z = c%Z_index
        RetVal = 0

        IF(x>0) RetVal = RetVal + 1
        IF(x<UBOUND(Cells,1)) RetVal = RetVal + 1
        IF(y>0) RetVal = RetVal + 1
        IF(y<UBOUND(Cells,2)) RetVal = RetVal + 1
        IF(z>0) RetVal = RetVal + 1
        IF(z<UBOUND(Cells,3)) RetVal = RetVal + 1

        RETURN

    END FUNCTION

    !Use GetAvailableNeighborCountForThisCell to first get the size of the array coming back!
    FUNCTION GetAvailableNeighborsForThisCell(c) RESULT (RetVal)

        TYPE(CartesianCell), INTENT(IN) :: c
        INTEGER, ALLOCATABLE, DIMENSION(:) :: RetVal

        INTEGER :: x, y, z
        INTEGER :: Ctr

        x = c%X_index
        y = c%Y_index
        z = c%Z_index
        RetVal = 0
        Ctr = -1

        IF(x>0) THEN
            Ctr = Ctr + 1
            RetVal(Ctr) = Direction_NegativeX
        END IF
        IF(x<UBOUND(Cells,1)) THEN
            Ctr = Ctr + 1
            RetVal(Ctr) = Direction_PositiveX
        END IF
        IF(y>0) THEN
            Ctr = Ctr + 1
            RetVal(Ctr) = Direction_NegativeY
        END IF
        IF(y<UBOUND(Cells,2)) THEN
            Ctr = Ctr + 1
            RetVal(Ctr) = Direction_PositiveY
        END IF
        IF(z>0) THEN
            Ctr = Ctr + 1
            RetVal(Ctr) = Direction_NegativeZ
        END IF
        IF(z<UBOUND(Cells,3)) THEN
            Ctr = Ctr + 1
            RetVal(Ctr) = Direction_PositiveZ
        END IF

        RETURN

    END FUNCTION

END MODULE
```

Listing B.3: Standalone Ground Heat Exchanger Model Source: Main Routines

```fortran
MODULE PiechowskiInputManager

    USE PiechowskiData
    USE Sim
    USE Extensions
    IMPLICIT NONE
    PUBLIC

    TYPE DirectionPackage
        REAL(r64) :: ExtentMax
        INTEGER :: Direction !From Enum: RegionType
        CHARACTER(len=1) :: ID
    END TYPE

    PUBLIC PerformInputProcessing

    CONTAINS

    TYPE(DirectionPackage) FUNCTION DirectionPack(dir) RESULT(RetVal)
        INTEGER, INTENT(IN) :: dir ! From Enum: RegionType
        SELECT CASE (dir)
        CASE (RegionType_XDirection)
            RetVal = DirectionPackage(Extents%XMax, RegionType_XDirection, 'X')
        CASE (RegionType_YDirection)
            RetVal = DirectionPackage(Extents%YMax, RegionType_YDirection, 'Y')
        CASE (RegionType_ZDirection)
            RetVal = DirectionPackage(Extents%ZMax, RegionType_ZDirection, 'Z')
        CASE DEFAULT
            !Debug error
        END SELECT
        RETURN
    END FUNCTION

    SUBROUTINE PerformInputProcessing()

        !fill the data structure
        CALL InputProcessor()

        !from the input data structure, generate and initialize the mesh
        CALL DevelopMesh()

        !initialize cell array
        CALL ResetAllCellTempTypesToDefault()

    END SUBROUTINE

    SUBROUTINE InputProcessor()

        LOGICAL :: MeshCountInfoAlreadyShown
        INTEGER :: PipeCtr

        INTEGER, PARAMETER :: FileUnit = 123
        LOGICAL :: ParameterFileExists
        INTEGER :: IOStatus
        CHARACTER(len=100) :: ReadLine
        CHARACTER(len=30) :: Key
        REAL(r64) :: Value
        INTEGER :: Pos

        !Come up with some default values first
        REAL(r64) :: KusudaAvgTemp = 15.5 !C
        REAL(r64) :: KusudaAvgAmp  = 12.8 !C
        REAL(r64) :: KusudaPhase   = 17.3 !days

        REAL(r64) :: GroundDensity = 962.0 !kg/m3
        REAL(r64) :: GroundSpecHeat = 2576.0 !J/kg-K

        !Then read the input file to override as necessary
        INQUIRE(FILE='parameters', EXIST=ParameterFileExists)
        IF (ParameterFileExists) THEN

            OPEN (FileUnit, FILE='parameters')

            DO !Indefinitely

                ! read the entire line from the data file
                READ(FileUnit, FMT='(A)', IOSTAT=IOStatus) ReadLine

                ! if it is a comment line then skip it
                IF (ReadLine(1:1)=='!') CYCLE

                ! if there is a problem then exit
                IF (IOStatus .NE. 0) EXIT

                ! check the Key and assign the value
                ReadLine = ADJUSTL(ReadLine)
                Pos = SCAN(ReadLine, '=')
                IF (Pos <= 1) THEN
                    !bad line?
                END IF

                READ(ReadLine(1:Pos-1), *) Key
                READ(ReadLine(Pos+1:), *) Value
                CALL To_upper(Key)
                SELECT CASE (TRIM(Key))
                CASE ('KUSUDAAVGTEMP')
                    KusudaAvgTemp = Value
                    WRITE (*, *) '␣*␣Override␣*␣KusudaAvgTemp␣␣=␣', KusudaAvgTemp
                CASE ('KUSUDAAVGAMP')
                    KusudaAvgAmp = Value
                    WRITE (*, *) '␣*␣Override␣*␣KusudaAvgAmp␣␣␣=␣', KusudaAvgAmp
                CASE ('KUSUDAPHASE')
                    KusudaPhase = Value
                    WRITE (*, *) '␣*␣Override␣*␣KusudaPhase␣␣␣␣=␣', KusudaPhase
                CASE ('GROUNDDENSITY')
                    GroundDensity = Value
                    WRITE (*, *) '␣*␣Override␣*␣GroundDensity␣␣=␣', GroundDensity
                CASE ('GROUNDSPECHEAT')
                    GroundSpecHeat = Value
                    WRITE (*, *) '␣*␣Override␣*␣GroundSpecHeat␣=␣', GroundSpecHeat
                CASE DEFAULT

                END SELECT

            END DO

            CLOSE(FileUnit)

        END IF

        Extents%Xmax = 10
        Extents%Ymax = 4.877 !16 feet
        Extents%Zmax = 36.84

        CALL ReadPipeCircuitData()
        IF (Has%PipeCircuit) THEN
            !' then some general pipe properties
            PipeCircuit%PipeSize%InnerDia = 0.016d0
            PipeCircuit%PipeSize%OuterDia = 0.02667d0
            !' then some general insulation properties
            !Read a flag whether or not there is insulation
            IF (.FALSE.) THEN
                PipeCircuit%InsulationSize%OuterDia = 0.011d0
                PipeCircuit%InsulationSize%InnerDia = PipeCircuit%PipeSize%OuterDia
                Has%Insulation = .TRUE.
            ELSE
                Has%Insulation = .FALSE.
            END IF
            !' wait to evaluate the pipe locations until we read in basement data to see if they are shifted...
        END IF


        !' next meshing properties
        MeshCountInfoAlreadyShown = .FALSE.
        !' first x values
        Mesh%X%MeshDistribution = MeshDistribution_Uniform
        Mesh%X%RegionMeshCount  = 9
        IF (Mesh%X%MeshDistribution == MeshDistribution_SymmetricGeometric) THEN
            IF (MOD(Mesh%X%RegionMeshCount, 2) .NE. 0) THEN
                Mesh%X%RegionMeshCount = Mesh%X%RegionMeshCount + 1
                MeshCountInfoAlreadyShown = .TRUE.
                Mesh%X%GeometricSeriesCoefficient = 1.4d0
            ELSE
                Mesh%X%GeometricSeriesCoefficient = 1.0d0
            END IF
        END IF

        Mesh%Y%MeshDistribution = MeshDistribution_Uniform
        Mesh%Y%RegionMeshCount  = 32
        IF (Mesh%Y%MeshDistribution == MeshDistribution_SymmetricGeometric) THEN
            IF (MOD(Mesh%Y%RegionMeshCount, 2) .NE. 0) THEN
                Mesh%Y%RegionMeshCount = Mesh%Y%RegionMeshCount + 1
                MeshCountInfoAlreadyShown = .TRUE.
                Mesh%Y%GeometricSeriesCoefficient = 1.4d0
            ELSE
                Mesh%Y%GeometricSeriesCoefficient = 1.0d0
            END IF
        END IF

        Mesh%Z%MeshDistribution = MeshDistribution_Uniform
        Mesh%Z%RegionMeshCount  = 9
        Mesh%Z%BoundaryType = BoundaryType_Farfield
        IF (Mesh%Z%MeshDistribution == MeshDistribution_SymmetricGeometric) THEN
            IF (MOD(Mesh%Z%RegionMeshCount, 2) .NE. 0) THEN
                Mesh%Z%RegionMeshCount = Mesh%Z%RegionMeshCount + 1
                MeshCountInfoAlreadyShown = .TRUE.
                Mesh%Z%GeometricSeriesCoefficient = 1.1d0
            ELSE
                Mesh%Z%GeometricSeriesCoefficient = 1.0d0
            END IF
        END IF
```

```
      IF (Has%PipeCircuit) THEN
          Mesh%Radial%NumRadialCells = 2
          Mesh%Radial%RadialMeshThickness = 0.03
      END IF

      IF (Has%PipeCircuit) THEN
          PipeProperties%Conductivity = 0.3895
          PipeProperties%Density = 641
          PipeProperties%SpecificHeat = 2405
      END IF

      GroundProperties%Conductivity = 1.08
      GroundProperties%Density = GroundDensity
      GroundProperties%SpecificHeat = GroundSpecHeat

      IF (Has%PipeCircuit .AND. Has%Insulation) THEN
          InsulationProperties%Conductivity = 0.154
          InsulationProperties%Density = 500
          InsulationProperties%SpecificHeat = 816
      END IF

      Farfield%Model = FarfieldModel_KusudaAchenbach
      SELECT CASE (Farfield%Model)
      CASE (FarfieldModel_Constant)
          Farfield%Constant%Temperature = 12
      CASE (FarfieldModel_ConstantLinear)
          Farfield%ConstantLinear%SurfaceTemperature = 12
          Farfield%ConstantLinear%Slope = -2
      CASE (FarfieldModel_KusudaAchenbach)
          Farfield%KusudaAchenbach% AverageGroundTemperature = KusudaAvgTemp
          Farfield%KusudaAchenbach% AverageGroundTemperatureAmplitude = KusudaAvgAmp
          Farfield%KusudaAchenbach% PhaseShiftOfMinGroundTempDays = KusudaPhase
          Farfield%KusudaAchenbach% PhaseShiftOfMinGroundTemp = Farfield%KusudaAchenbach%
               PhaseShiftOfMinGroundTempDays * SecondsInDay
      END SELECT

      !'basement zone boundary data
      BasementZone%Depth = 2.5d0
      BasementZone%Width = 6.0d0
      IF ((BasementZone%Depth > 0) .OR. (BasementZone%Width > 0)) THEN
          Has%Basement = .FALSE.
          BasementZone%ShiftPipesByWidth = .TRUE.
          IF (BasementZone%ShiftPipesByWidth) THEN
              IF (Has%PipeCircuit) THEN
                  DO PipeCtr = LBOUND(PipeCircuit%PipeSegments, 1), UBOUND(PipeCircuit%PipeSegments, 1)
                      PipeCircuit%PipeSegments(PipeCtr)%PipeLocation%X = PipeCircuit%PipeSegments(PipeCtr)%
                           PipeLocation%X + BasementZone%Width
                  END DO
              END IF
          END IF
          BasementZone%UnderBasementBoundaryType = BoundaryType_Adiabatic
          BasementZone%BasementWall%MyBase%Conductivity = 1
          BasementZone%BasementWall%MyBase%Density = 1
          BasementZone%BasementWall%MyBase%SpecificHeat = 1
          BasementZone%BasementWall%Thickness = 0.1
          BasementZone%BasementFloor%MyBase%Conductivity = 1
          BasementZone%BasementFloor%MyBase%Density = 1
          BasementZone%BasementFloor%MyBase%SpecificHeat = 1
          BasementZone%BasementFloor%Thickness = 0.1
      ELSE
          Has%Basement = .FALSE.
      END IF

      SimControls%DomainInitialTemperature = 10.0d0
      SimControls%Cartesian%Convergence_CurrentToPrevIteration = 0.00001d0
      SimControls%Cartesian%MaxIterationsPerTS = 250
      IF (Has%PipeCircuit) THEN
          SimControls%Radial%Convergence_CurrentToPrevIteration = 0.005
          SimControls%Radial%MaxIterationsPerTS = 150
      END IF

  END SUBROUTINE

SUBROUTINE To_upper(str)
  CHARACTER(*), INTENT(in out) :: str
  INTEGER :: i

  DO i = 1, len(str)
    SELECT CASE(str(i:i))
      CASE("a":"z")
        str(i:i) = ACHAR(IACHAR(str(i:i))-32)
    END SELECT
  END DO
END SUBROUTINE To_upper

  SUBROUTINE ReadPipeCircuitData()

      LOGICAL :: TrustMeItHasOne = .FALSE.
      INTEGER :: TrustMeNumPipes = 6
      INTEGER :: PipeSegmentCounter
      REAL(r64) :: XCoordinate
      REAL(r64) :: YCoordinate
      IF (TrustMeItHasOne) THEN
          Has%PipeCircuit = .TRUE.
      ELSE
          Has%PipeCircuit = .FALSE.
          RETURN
      END IF

      ALLOCATE(PipeCircuit%PipeSegments(0:TrustMeNumPipes-1))

      PipeSegmentCounter = 0
      PipeCircuit%PipeSegments(PipeSegmentCounter)%FlowDirection = SegmentFlow_IncreasingZ
      PipeCircuit%PipeSegments(PipeSegmentCounter)%PipeLocation = PointF(0.67, Extents%Ymax - 2.20)

      PipeSegmentCounter = PipeSegmentCounter + 1
      PipeCircuit%PipeSegments(PipeSegmentCounter)%FlowDirection = SegmentFlow_IncreasingZ
      PipeCircuit%PipeSegments(PipeSegmentCounter)%PipeLocation = PointF(0.95, Extents%Ymax - 2.20)

      PipeSegmentCounter = PipeSegmentCounter + 1
      PipeCircuit%PipeSegments(PipeSegmentCounter)%FlowDirection = SegmentFlow_IncreasingZ
      PipeCircuit%PipeSegments(PipeSegmentCounter)%PipeLocation = PointF(1.23, Extents%Ymax - 2.20)

      PipeSegmentCounter = PipeSegmentCounter + 1
      PipeCircuit%PipeSegments(PipeSegmentCounter)%FlowDirection = SegmentFlow_DecreasingZ
      PipeCircuit%PipeSegments(PipeSegmentCounter)%PipeLocation = PointF(1.40, Extents%Ymax - 1.94)

      PipeSegmentCounter = PipeSegmentCounter + 1
      PipeCircuit%PipeSegments(PipeSegmentCounter)%FlowDirection = SegmentFlow_DecreasingZ
      PipeCircuit%PipeSegments(PipeSegmentCounter)%PipeLocation = PointF(1.40, Extents%Ymax - 1.66)

      PipeSegmentCounter = PipeSegmentCounter + 1
      PipeCircuit%PipeSegments(PipeSegmentCounter)%FlowDirection = SegmentFlow_DecreasingZ
      PipeCircuit%PipeSegments(PipeSegmentCounter)%PipeLocation = PointF(1.40, Extents%Ymax - 1.39)


      RETURN

  END SUBROUTINE

  ! ==================================================
  ! =========== Mesh Development routines ============
  ! ==================================================

  SUBROUTINE DevelopMesh()

      TYPE(GridRegion), ALLOCATABLE, DIMENSION(:) :: XPartitionRegions
      TYPE(GridRegion), ALLOCATABLE, DIMENSION(:) :: YPartitionRegions
      TYPE(GridRegion), ALLOCATABLE, DIMENSION(:) :: ZPartitionRegions
      TYPE(GridRegion), ALLOCATABLE, DIMENSION(:) :: XRegions
      TYPE(GridRegion), ALLOCATABLE, DIMENSION(:) :: YRegions
      TYPE(GridRegion), ALLOCATABLE, DIMENSION(:) :: ZRegions
      REAL(r64),        ALLOCATABLE, DIMENSION(:) :: XBoundaryPoints
      REAL(r64),        ALLOCATABLE, DIMENSION(:) :: YBoundaryPoints
      REAL(r64),        ALLOCATABLE, DIMENSION(:) :: ZBoundaryPoints

      INTEGER :: RegionListCount
      INTEGER :: BoundaryListCount
      LOGICAL :: XPartitionsExist
      LOGICAL :: YPartitionsExist
      LOGICAL :: ZPartitionsExist

      !'****** LAYOUT PARTITIONS ******'
      CALL CreatePartitionCenterList()

      IF (ALLOCATED(Partitions%X)) THEN
          ALLOCATE(XPartitionRegions(0:UBOUND(Partitions%X,1)))
          XPartitionsExist = .TRUE.
      ELSE
          ALLOCATE(XPartitionRegions(0:0))
          ALLOCATE(Partitions%X(0:0))
          XPartitionsExist = .FALSE.
      END IF
      XPartitionRegions = CreatePartitionRegionList(Partitions%X, DirectionPack(RegionType_XDirection),
           XPartitionsExist, UBOUND(Partitions%X, 1))

      IF (ALLOCATED(Partitions%Y)) THEN
          ALLOCATE(YPartitionRegions(0:UBOUND(Partitions%Y,1)))
          YPartitionsExist = .TRUE.
      ELSE
          ALLOCATE(YPartitionRegions(0:0))
          ALLOCATE(Partitions%Y(0:0))
          YPartitionsExist = .FALSE.
      END IF
      YPartitionRegions = CreatePartitionRegionList(Partitions%Y, DirectionPack(RegionType_YDirection),
           YPartitionsExist, UBOUND(Partitions%Y, 1))

      ZPartitionsExist = .FALSE.

      !'***** LAYOUT MESH REGIONS *****'
      RegionListCount = CreateRegionListCount(XPartitionRegions, DirectionPack(RegionType_XDirection),
           XPartitionsExist)
      ALLOCATE(XRegions(0:RegionListCount-1))
      XRegions = CreateRegionList(XPartitionRegions, DirectionPack(RegionType_XDirection), RegionListCount-1,
           XPartitionsExist, BasementWallXIndex=BasementWallXIndex)
      RegionListCount = CreateRegionListCount(YPartitionRegions, DirectionPack(RegionType_YDirection),
           YPartitionsExist)
      ALLOCATE(YRegions(0:RegionListCount-1))
      YRegions = CreateRegionList(YPartitionRegions, DirectionPack(RegionType_YDirection), RegionListCount-1,
           YPartitionsExist, BasementFloorYIndex=BasementFloorYIndex)

      RegionListCount = CreateRegionListCount(ZPartitionRegions, DirectionPack(RegionType_ZDirection),
           ZPartitionsExist)
      ALLOCATE(ZRegions(0:RegionListCount-1))
      ZRegions = CreateRegionList(ZPartitionRegions, DirectionPack(RegionType_ZDirection), RegionListCount-1,
           ZPartitionsExist)

      !'** MAKE REGIONS > BOUNDARIES **'
      BoundaryListCount = CreateBoundaryListCount(XRegions, DirectionPack(RegionTYpe_XDirection))
      ALLOCATE(XBoundaryPoints(0:BoundaryListCount-1))
      XBoundaryPoints = CreateBoundaryList(XRegions, DirectionPack(RegionTYpe_XDirection), 0, BoundaryListCount
           -1)

      BoundaryListCount = CreateBoundaryListCount(YRegions, DirectionPack(RegionTYpe_YDirection))
      ALLOCATE(YBoundaryPoints(0:BoundaryListCount-1))
      YBoundaryPoints = CreateBoundaryList(YRegions, DirectionPack(RegionTYpe_YDirection), 0, BoundaryListCount
           -1)

      BoundaryListCount = CreateBoundaryListCount(ZRegions, DirectionPack(RegionTYpe_ZDirection))
      ALLOCATE(ZBoundaryPoints(0:BoundaryListCount-1))
      ZBoundaryPoints = CreateBoundaryList(ZRegions, DirectionPack(RegionTYpe_ZDirection), 0, BoundaryListCount
           -1)

      !'****** DEVELOP CELL ARRAY *****'
      CALL CreateCellArray(XBoundaryPoints, YBoundaryPoints, ZBoundaryPoints, BasementWallXIndex,
           BasementFloorYIndex)

      !'***** SETUP CELL NEIGHBORS ****'
      CALL SetupCellNeighbors()

      !'** SET UP PIPE CIRCUIT CELLS **'
      IF (Has%PipeCircuit) CALL SetupPipeCircuitInOutCells()

      IF(ALLOCATED(XPartitionRegions)) DEALLOCATE(XPartitionRegions)
      IF(ALLOCATED(YPartitionRegions)) DEALLOCATE(YPartitionRegions)
      IF(ALLOCATED(ZPartitionRegions)) DEALLOCATE(ZPartitionRegions)
      IF(ALLOCATED(XRegions)) DEALLOCATE(XRegions)
      IF(ALLOCATED(YRegions)) DEALLOCATE(YRegions)
      IF(ALLOCATED(ZRegions)) DEALLOCATE(ZRegions)
      IF(ALLOCATED(XBoundaryPoints)) DEALLOCATE(XBoundaryPoints)
      IF(ALLOCATED(YBoundaryPoints)) DEALLOCATE(YBoundaryPoints)
      IF(ALLOCATED(ZBoundaryPoints)) DEALLOCATE(ZBoundaryPoints)

      RETURN

  END SUBROUTINE

  SUBROUTINE CreatePartitionCenterList()

      REAL(r64) :: BasementDistFromBottom
      INTEGER :: PipeCtr
      INTEGER :: PreviousUbound
      TYPE(MeshPartition), ALLOCATABLE, DIMENSION(:) :: PreviousEntries
      REAL(r64) :: PipeCellWidth
      REAL(r64) :: SurfCellWidth !Basement surface...

      !the fraction of domain extent to use for the basement cells
      !actual dimension shouldn't matter for calculation purposes
      REAL(r64), PARAMETER :: BasementCellFraction = 0.001d0


      IF (Has%PipeCircuit) THEN

          IF (.NOT. Has%Insulation) THEN
              PipeCellWidth = PipeCircuit%PipeSize%OuterDia
          ELSE
              PipeCellWidth = PipeCircuit%InsulationSize%OuterDia
          END IF
          PipeCellWidth = PipeCellWidth + 2 * Mesh%Radial%RadialMeshThickness

          !'NOTE: pipe location y values have already been corrected to be measured from the bottom surface
          !'in input they are measured by depth, but internally they are referred to by distance from y = 0, or
               the bottom boundary
          DO PipeCtr = LBOUND(PipeCircuit%PipeSegments, 1), UBOUND(PipeCircuit%PipeSegments, 1)
              IF (.NOT. ALLOCATED(Partitions%X)) THEN
                  ALLOCATE(Partitions%X(0:0))
                  Partitions%X(0) = MeshPartition(PipeCircuit%PipeSegments(PipeCtr)%PipeLocation%X,
                       PartitionType_Pipe, PipeCellWidth)
              ELSEIF (.NOT. MeshPartitionArray_Contains(Partitions%X, PipeCircuit%PipeSegments(PipeCtr)%
                   PipeLocation%X)) THEN
                  PreviousUbound = UBOUND(Partitions%X, 1)
                  IF (ALLOCATED(PreviousEntries)) DEALLOCATE(PreviousEntries)
                  ALLOCATE(PreviousEntries(0:PreviousUbound))
                  PreviousEntries = Partitions%X
                  DEALLOCATE(Partitions%X)
                  ALLOCATE(Partitions%X(0:PreviousUbound+1))
                  Partitions%X(0:PreviousUbound) = PreviousEntries
                  Partitions%X(PreviousUbound + 1) = MeshPartition(PipeCircuit%PipeSegments(PipeCtr)%
                       PipeLocation%X, PartitionType_Pipe, PipeCellWidth)
              END IF
              IF (.NOT. ALLOCATED(Partitions%Y)) THEN
                  ALLOCATE(Partitions%Y(0:0))
                  Partitions%Y(0) = MeshPartition(PipeCircuit%PipeSegments(PipeCtr)%PipeLocation%Y,
                       PartitionType_Pipe, PipeCellWidth)
              ELSEIF (.NOT. MeshPartitionArray_Contains(Partitions%Y, PipeCircuit%PipeSegments(PipeCtr)%
                   PipeLocation%Y)) THEN
                  PreviousUbound = UBOUND(Partitions%Y, 1)
                  IF (ALLOCATED(PreviousEntries)) DEALLOCATE(PreviousEntries)
                  ALLOCATE(PreviousEntries(0:PreviousUbound))
                  PreviousEntries = Partitions%Y
                  DEALLOCATE(Partitions%Y)
                  ALLOCATE(Partitions%Y(0:PreviousUbound+1))
                  Partitions%Y(0:PreviousUbound) = PreviousEntries
                  Partitions%Y(PreviousUbound + 1) = MeshPartition(PipeCircuit%PipeSegments(PipeCtr)%
                       PipeLocation%Y, PartitionType_Pipe, PipeCellWidth)
              END IF
          END DO
      END IF

      IF (Has%Basement) THEN
          !'NOTE: the basement depth is still a depth from the ground surface, need to correct this here
          IF (BasementZone%Width > 0) THEN
              SurfCellWidth = Extents%Xmax * BasementCellFraction
              IF (.NOT. ALLOCATED(Partitions%X)) THEN
                  ALLOCATE(Partitions%X(0:0))
                  Partitions%X(0) = MeshPartition(BasementZone%Width, PartitionType_BasementWall, SurfCellWidth
                       )
              ELSEIF (.NOT. MeshPartitionArray_Contains(Partitions%X, BasementZone%Width)) THEN
                  PreviousUbound = UBOUND(Partitions%X, 1)
                  IF (ALLOCATED(PreviousEntries)) DEALLOCATE(PreviousEntries)
                  ALLOCATE(PreviousEntries(0:PreviousUbound))
                  PreviousEntries = Partitions%X
                  DEALLOCATE(Partitions%X)
                  ALLOCATE(Partitions%X(0:PreviousUbound+1))
                  Partitions%X(0:PreviousUbound) = PreviousEntries
                  Partitions%X(PreviousUbound + 1) = MeshPartition(BasementZone%Width,
                       PartitionType_BasementWall, SurfCellWidth)
              END IF
          END IF
          IF (BasementZone%Depth > 0) THEN
              SurfCellWidth = Extents%Ymax * BasementCellFraction
              BasementDistFromBottom = Extents%Ymax - BasementZone%Depth
              IF (.NOT. ALLOCATED(Partitions%Y)) THEN
                  ALLOCATE(Partitions%Y(0:0))
                  Partitions%Y(0) = MeshPartition(BasementDistFromBottom, PartitionType_BasementFloor,
                       SurfCellWidth)
              ELSEIF (.NOT. MeshPartitionArray_Contains(Partitions%Y, BasementDistFromBottom)) THEN
                  PreviousUbound = UBOUND(Partitions%Y, 1)
                  IF (ALLOCATED(PreviousEntries)) DEALLOCATE(PreviousEntries)
                  ALLOCATE(PreviousEntries(0:PreviousUbound))
                  PreviousEntries = Partitions%Y
                  DEALLOCATE(Partitions%Y)
                  ALLOCATE(Partitions%Y(0:PreviousUbound+1))
                  Partitions%Y(0:PreviousUbound) = PreviousEntries
                  Partitions%Y(PreviousUbound + 1) = MeshPartition(BasementDistFromBottom,
                       PartitionType_BasementFloor, SurfCellWidth)
              END IF
          END IF
      END IF

      CALL MeshPartition_SelectionSort(Partitions%X)
      CALL MeshPartition_SelectionSort(Partitions%Y)

  END SUBROUTINE

  FUNCTION CreatePartitionRegionList(ThesePartitionCenters, Dir, PartitionsExist, PartitionsUBound) RESULT(
       ThesePartitionRegions)

      TYPE(MeshPartition), ALLOCATABLE, DIMENSION(:), INTENT(IN) :: ThesePartitionCenters
      TYPE(DirectionPackage), INTENT(IN) :: Dir
      INTEGER, INTENT(IN) :: PartitionsUbound
      LOGICAL, INTENT(IN) :: PartitionsExist

      TYPE(GridRegion), DIMENSION(0:PartitionsUBound) :: ThesePartitionRegions

      REAL(r64) :: PipeCellWidthBy2
      REAL(r64) :: PipeCellWidth
      INTEGER :: Index
      REAL(r64) :: ThisCellWidthBy2
      INTEGER :: ThisPartitionType !From Enum: RegionType
      REAL(r64) :: CellLeft
      REAL(r64) :: CellRight
      INTEGER :: PreviousUbound
      TYPE(GridRegion), ALLOCATABLE, DIMENSION(:) :: PreviousEntries

      IF (.NOT. PartitionsExist) THEN
          RETURN
      END IF

      !'loop across all partitions
      DO Index = LBOUND(ThesePartitionCenters, 1), UBOUND(ThesePartitionCenters, 1)

          !retrieve a cell half-width and a partition type
          ThisCellWidthBy2 = ThesePartitionCenters(Index)%TotalWidth / 2.0d0
          ThisPartitionType = ThesePartitionCenters(Index)%PartitionType

          !'use this half width to validate the region and add it to the collection
          CellLeft = ThesePartitionCenters(Index)%rDimension - ThisCellWidthBy2
          CellRight = ThesePartitionCenters(Index)%rDimension + ThisCellWidthBy2

          ThesePartitionRegions(Index)%Min = CellLeft
          ThesePartitionRegions(Index)%Max = CellRight

          !Need to map partition type into region type parameters, since they are different enumerations
          SELECT CASE (ThisPartitionType)
          CASE (PartitionType_BasementWall)
              ThesePartitionRegions(Index)%RegionType = RegionType_BasementWall
          CASE (PartitionType_BasementFloor)
              ThesePartitionRegions(Index)%RegionType = RegionType_BasementFloor
          CASE (PartitionType_Pipe)
              ThesePartitionRegions(Index)%RegionType = RegionType_Pipe
          CASE DEFAULT
              !diagnostic error
          END SELECT

      END DO

      RETURN

  END FUNCTION

  INTEGER FUNCTION CreateRegionListCount(ThesePartitionRegions, Dir, PartitionsExist) RESULT(RetVal)

      TYPE(GridRegion), ALLOCATABLE, DIMENSION(:), INTENT(IN) :: ThesePartitionRegions
      TYPE(DirectionPackage), INTENT(IN) :: Dir
      LOGICAL, INTENT(IN) :: PartitionsExist

      INTEGER :: Index

      RetVal = 0
      IF (PartitionsExist) THEN
          DO Index = LBOUND(ThesePartitionRegions,1), UBOUND(ThesePartitionRegions,1)
              !'add a mesh region to the "left" of the partition
              RetVal = RetVal + 1
              !'then add the pipe node itself
              RetVal = RetVal + 1
              !some cleanup based on where we are
              IF ((Index==0 .AND. SIZE(ThesePartitionRegions)==1) .OR. &
                  (Index == UBOUND(ThesePartitionRegions,1) .AND. ThesePartitionRegions(Index)%Max < Dir%
                      ExtentMax)) THEN
                  !'if there is only one partition, add a mesh region to the "right" before we leave
                  !'or if we are on the last partition, and we have room on the "right" side then add a mesh
                      region
                  RetVal = RetVal + 1
              END IF
          END DO
      ELSE !Input partitions were not allocate
          !'if we don't have a region, we still need to make a single mesh region
          RetVal = RetVal + 1
      END IF

      RETURN

  END FUNCTION


  FUNCTION CreateRegionList(ThesePartitionRegions, Dir, RetValUbound, PartitionsExist, BasementWallXIndex,
         BasementFloorYIndex) RESULT(RetVal)

      TYPE(GridRegion), ALLOCATABLE, DIMENSION(:), INTENT(IN) :: ThesePartitionRegions
      TYPE(DirectionPackage), INTENT(IN) :: Dir
      INTEGER, INTENT(IN) :: RetValUBound
      LOGICAL, INTENT(IN) :: PartitionsExist
      INTEGER, INTENT(IN OUT), OPTIONAL :: BasementWallXIndex
      INTEGER, INTENT(IN OUT), OPTIONAL :: BasementFloorYIndex

      TYPE(GridRegion), DIMENSION(0:RetValUbound) :: RetVal
      TYPE(TempGridRegionData), DIMENSION(0:RetValUbound) :: TempRegions

      TYPE(GridRegion) :: ThisRegion
      TYPE(GridRegion) :: TempRegion
      TYPE(TempGridRegionData) :: PreviousRegion
      REAL(r64) :: LeftRegionExtent
      INTEGER :: PreviousUbound
      INTEGER :: Index
      INTEGER :: SubIndex
      INTEGER :: CellCountUpToNow
      INTEGER :: NumCellWidths

      PreviousUbound = -1
      IF (PartitionsExist) THEN
          DO Index = LBOUND(ThesePartitionRegions,1), UBOUND(ThesePartitionRegions,1)
              ThisRegion = ThesePartitionRegions(Index)
```

```
                IF (Index == 0) THEN
                    LeftRegionExtent = 0.0
                ELSE
                    LeftRegionExtent = ThesePartitionRegions(Index - 1)%Max
                END IF
                !'add a mesh region to the "left" of the partition
                PreviousUbound = PreviousUbound + 1
                TempRegions(PreviousUbound) = TempGridRegionData(LeftRegionExtent, ThisRegion%Min, Dir%Direction)

                !'alert calling routines to the location of the basement cells within the domain
                CellCountUpToNow = 0
                DO SubIndex = LBOUND(TempRegions,1), PreviousUbound
                    PreviousRegion = TempRegions(SubIndex)
                    SELECT CASE (PreviousRegion%RegionType)
                    CASE (RegionType_Pipe, RegionType_BasementFloor, RegionType_BasementWall)
                        CellCountUpToNow = CellCountUpToNow + 1
                    CASE DEFAULT
                        CellCountUpToNow = CellCountUpToNow + GetCellWidthsCount(Dir%Direction)
                    END SELECT
                END DO

                IF (ThisRegion%RegionType == RegionType_BasementWall) THEN
                    IF (PRESENT(BasementWallXIndex)) BasementWallXIndex = CellCountUpToNow
                ELSEIF (ThisRegion%RegionType == RegionType_BasementFloor) THEN
                    IF (PRESENT(BasementFloorYIndex)) BasementFloorYIndex = CellCountUpToNow
                END IF

                !'then add the pipe node itself
                PreviousUbound = PreviousUbound + 1
                TempRegions(PreviousUbound) = TempGridRegionData(ThisRegion%Min, ThisRegion%Max, ThisRegion%
                     RegionType)

                !some cleanup based on where we are
                IF ((Index==0 .AND. SIZE(ThesePartitionRegions)==1) .OR. &
                    (Index == UBOUND(ThesePartitionRegions,1) .AND. ThisRegion%Max < Dir%ExtentMax)) THEN
                    !'if there is only one partition, add a mesh region to the "right" before we leave
                    !'or if we are on the last partition, and we have room on the "right" side then add a mesh
                         region
                    PreviousUbound = PreviousUbound + 1
                    TempRegions(PreviousUbound) = TempGridRegionData(ThisRegion%Max, Dir%ExtentMax, Dir%Direction
                         )
                END IF
            END DO
        ELSE !Input partitions were not allocate
            !'if we don't have a region, we still need to make a single mesh region
            TempRegions(0) = TempGridRegionData(0.0, Dir%ExtentMax, Dir%Direction)
        END IF

        !'finally repackage the grid regions into the final class form with cell counts included
        DO Index = LBOUND(TempRegions,1), UBOUND(TempRegions,1)
            RetVal(Index)%Min = TempRegions(Index)%Min
            RetVal(Index)%Max = TempRegions(Index)%Max
            RetVal(Index)%RegionType = TempRegions(Index)%RegionType
            NumCellWidths = GetCellWidthsCount(Dir%Direction)
            IF (ALLOCATED(RetVal(Index)%CellWidths)) DEALLOCATE(RetVal(Index)%CellWidths)
            ALLOCATE (RetVal(Index)%CellWidths(0:NumCellWidths-1))
            CALL GetCellWidths(RetVal(Index))
        END DO

        RETURN

    END FUNCTION

    INTEGER FUNCTION CreateBoundaryListCount(RegionList, dir) RESULT(RetVal)

        TYPE(GridRegion), ALLOCATABLE, DIMENSION(:), INTENT(IN) :: RegionList
        TYPE(DirectionPackage) :: dir

        INTEGER :: Index
        INTEGER :: CellWidthCtr

        RetVal = 0

        DO Index = LBOUND(RegionList,1), UBOUND(RegionList,1)
            SELECT CASE (RegionList(Index)%RegionType)
            CASE (RegionType_Pipe, RegionType_BasementFloor, RegionType_BasementWall)
                RetVal = RetVal + 1
            CASE DEFAULT
                IF (RegionList(Index)%RegionType == dir%Direction) THEN
                    DO CellWidthCtr = LBOUND(RegionList(Index)%CellWidths,1), UBOUND(RegionList(Index)%CellWidths
                         ,1)
                        RetVal = RetVal + 1
                    END DO
                END IF
            END SELECT
        END DO
        RetVal = RetVal + 1

        RETURN

    END FUNCTION

    FUNCTION CreateBoundaryList(RegionList, dir, RetValLbound, RetValUbound) RESULT(RetVal)
        TYPE(GridRegion), ALLOCATABLE, DIMENSION(:), INTENT(IN) :: RegionList
        TYPE(DirectionPackage) :: dir
        INTEGER, INTENT(IN) :: RetValLbound
        INTEGER, INTENT(IN) :: RetValUbound

        REAL(r64) :: RetVal(RetValLbound:RetValUbound)

        REAL(r64) :: StartingPointCounter
        INTEGER :: Index
        INTEGER :: Counter
        INTEGER :: CellWidthCtr

        Counter = -1
        DO Index = LBOUND(RegionList,1), UBOUND(RegionList,1)
            SELECT CASE (RegionList(Index)%RegionType)
            CASE (RegionType_Pipe, RegionType_BasementFloor, RegionType_BasementWall)
                Counter = Counter + 1
                RetVal(Counter) = RegionList(Index)%Min
            CASE DEFAULT
                IF (RegionList(Index)%RegionType == dir%Direction) THEN
                    StartingPointCounter = RegionList(Index)%Min
                    DO CellWidthCtr = LBOUND(RegionList(Index)%CellWidths,1), UBOUND(RegionList(Index)%CellWidths
                         ,1)
                        Counter = Counter + 1
                        RetVal(Counter) = StartingPointCounter
                        StartingPointCounter = StartingPointCounter + RegionList(Index)%CellWidths(CellWidthCtr)
                    END DO
                END IF
            END SELECT
        END DO
        RetVal(UBOUND(RetVal,1)) = dir%ExtentMax

        RETURN

    END FUNCTION

    SUBROUTINE CreateCellArray(XBoundaryPoints, YBoundaryPoints, ZBoundaryPoints, MaxBasementXNodeIndex,
         MinBasementYNodeIndex)

        TYPE tCellExtents
            TYPE(MeshExtents) :: MyBase
            REAL(r64)         :: Xmin
            REAL(r64)         :: Ymin
            REAL(r64)         :: Zmin
        END TYPE tCellExtents

        REAL(r64), ALLOCATABLE, DIMENSION(:), INTENT(IN) :: XBoundaryPoints
        REAL(r64), ALLOCATABLE, DIMENSION(:), INTENT(IN) :: YBoundaryPoints
        REAL(r64), ALLOCATABLE, DIMENSION(:), INTENT(IN) :: ZBoundaryPoints
        INTEGER, INTENT(IN) :: MaxBasementXNodeIndex
        INTEGER, INTENT(IN) :: MinBasementYNodeIndex

        INTEGER :: YIndexMax
        TYPE(DomainRectangle) :: BasementRectangle
        TYPE(tCellExtents) :: CellExtents
        TYPE(Point3DReal) :: Centroid
        TYPE(Point3DInteger) :: CellIndeces
        TYPE(RectangleF) :: XYRectangle
        INTEGER :: CellType !From Enum: CellType
        INTEGER :: ZWallCellType !From Enum: CellType
        INTEGER :: UnderBasementBoundary !From Enum: CellType
        INTEGER :: PipeCounter

        INTEGER   :: X
        INTEGER   :: CellXIndex
        REAL(r64) :: CellXMinValue
        REAL(r64) :: CellXMaxValue
        REAL(r64) :: CellXCenter
        REAL(r64) :: CellWidth

        INTEGER   :: Y
        INTEGER   :: CellYIndex
        REAL(r64) :: CellYMinValue
        REAL(r64) :: CellYMaxValue
        REAL(r64) :: CellYCenter
        REAL(r64) :: CellHeight

        INTEGER   :: Z
        INTEGER   :: CellZIndex
        REAL(r64) :: CellZMinValue
        REAL(r64) :: CellZMaxValue
        REAL(r64) :: CellZCenter
        REAL(r64) :: CellDepth

        INTEGER :: PipeIndex
        INTEGER :: NumRadialCells
        REAL(r64) :: InsulationThickness

        !'subtract 2 in each dimension:
        !'     one for zero based array
        !'     one because the boundary points contain one entry more than the number of cells WITHIN the domain
        ALLOCATE(Cells(0:SIZE(XBoundaryPoints) - 2, 0:SIZE(YBoundaryPoints) - 2, 0:SIZE(ZBoundaryPoints) - 2))
```

```
YIndexMax = UBOUND(Cells, 2)
BasementRectangle = DomainRectangle(0, MaxBasementXNodeIndex, MinBasementYNodeIndex, YIndexMax)

DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
    DO Y = LBOUND(Cells,2), UBOUND(Cells,2)
        DO X = LBOUND(Cells,1), UBOUND(Cells,1)

            !'set up x-direction variables
            CellXIndex = X  !'zero based index
            CellXMinValue = XBoundaryPoints(X)  !'left wall x-value
            CellXMaxValue = XBoundaryPoints(X + 1)  !'right wall x-value
            CellXCenter = (CellXMinValue + CellXMaxValue) / 2
            CellWidth = CellXMaxValue - CellXMinValue

            !'set up y-direction variables
            CellYIndex = Y !'zero based index
            CellYMinValue = YBoundaryPoints(Y)  !'bottom wall y-value
            CellYMaxValue = YBoundaryPoints(Y + 1)  !'top wall y-value
            CellYCenter = (CellYMinValue + CellYMaxValue) / 2
            CellHeight = CellYMaxValue - CellYMinValue

            !'set up z-direction variables
            CellZIndex = Z  !'zero based index
            CellZMinValue = ZBoundaryPoints(Z)  !'lower z value
            CellZMaxValue = ZBoundaryPoints(Z + 1)  !'higher z value
            CellZCenter = (CellZMinValue + CellZMaxValue) / 2
            CellDepth = CellZMaxValue - CellZMinValue

            !'set up an extent class for this cell
            CellExtents = tCellExtents(MeshExtents(CellXMaxValue, CellYMaxValue, CellZMaxValue),
                 CellXMinValue, CellYMinValue, CellZMinValue)

            !'set up centroid, index, and overall size
            Centroid = Point3DReal(CellXCenter, CellYCenter, CellZCenter)
            CellIndeces = Point3DInteger(CellXIndex, CellYIndex, CellZIndex)
            XYRectangle = RectangleF(CellXMinValue, CellYMinValue, CellWidth, CellHeight)

            !'determine cell type
            CellType = CellType_Unknown

            !'if this is a pipe node, some flags are needed
            PipeIndex = -1
            NumRadialCells = -1

            !'set up a z-pointer cell type variable
            IF (Mesh%Z%BoundaryType == BoundaryType_Adiabatic) THEN
                ZWallCellType = CellType_AdiabaticWall
            ELSEIF (Mesh%Z%BoundaryType == BoundaryType_Farfield) THEN
                ZWallCellType = CellType_FarfieldBoundary
            END IF

            IF (Has%Basement) THEN
                IF (BasementZone%UnderBasementBoundaryType == BoundaryType_Adiabatic) THEN
                    UnderBasementBoundary = CellType_AdiabaticWall
                ELSEIF (BasementZone%UnderBasementBoundaryType == BoundaryType_Farfield) THEN
                    UnderBasementBoundary = CellType_FarfieldBoundary
                END IF
            END IF

            !'apply boundary conditions
            IF (CellXIndex == MaxBasementXNodeIndex .AND. CellYIndex == MinBasementYNodeIndex) THEN
                CellType = CellType_BasementCorner
            ELSE IF (CellXIndex == MaxBasementXNodeIndex .AND. CellYIndex > MinBasementYNodeIndex) THEN
                CellType = CellType_BasementWall
            ELSE IF (CellXIndex < MaxBasementXNodeIndex .AND. CellYIndex == MinBasementYNodeIndex) THEN
                CellType = CellType_BasementFloor
            ELSE IF (CellXIndex < MaxBasementXNodeIndex .AND. CellYIndex > MinBasementYNodeIndex) THEN
                CellType = CellType_BasementCutAway
            ELSE IF (CellYIndex == UBOUND(Cells,2)) THEN
                CellType = CellType_GroundSurface
            ELSE IF (CellXIndex == 0) THEN
                IF (Has%Basement .AND. Y>0) THEN
                    CellType = UnderBasementBoundary !'this must come after the basement cutaway ELSEIF
                         branch
                ELSE
                    CellType = CellType_FarfieldBoundary
                END IF
            ELSE IF (CellXIndex == UBOUND(Cells,1) .OR. CellYIndex == 0) THEN
                CellType = CellType_FarfieldBoundary
            ELSE IF (CellZIndex == 0 .OR. CellZIndex == UBOUND(Cells,3)) THEN
                CellType = ZWallCellType
            END IF

            !'check to see if this is a pipe node...
            IF (Has%PipeCircuit) THEN
                DO PipeCounter = LBOUND(PipeCircuit%PipeSegments,1), UBOUND(PipeCircuit%PipeSegments,1)
                    IF (RectangleF_Contains(XYRectangle, PipeCircuit%PipeSegments(PipeCounter)%
                         PipeLocation)) THEN
                        !'inform the cell that it is a pipe node
                        CellType = CellType_Pipe
                        !'inform the cell of which pipe it contains
                        PipeIndex = PipeCounter
                        !'inform the pipe of what cell it is inside
                        CALL PipeSegmentInfo_InitPipeCells(PipeCircuit%PipeSegments(PipeCounter),
                             CellXIndex, CellYIndex)
                        !'set the number of cells to be generated in this near-pipe region
                        NumRadialCells = Mesh%Radial%NumRadialCells
                        !'exit the pipe counter loop
                        EXIT
                    END IF
                END DO
            END IF

            !'if it still isn't anything, then it is just an interior node
            IF (CellType == CellType_Unknown) THEN
                CellType = CellType_GeneralField
            END IF

            IF (Has%Insulation) THEN
                InsulationThickness = RadialSizing_Thickness(PipeCircuit%InsulationSize)
            END IF

            !'instantiate the cell class
            Cells(X, Y, Z)%X_min = CellExtents%Xmin
            Cells(X, Y, Z)%X_max = CellExtents%MyBase%Xmax
            Cells(X, Y, Z)%Y_min = CellExtents%Ymin
            Cells(X, Y, Z)%Y_max = CellExtents%MyBase%Ymax
            Cells(X, Y, Z)%Z_min = CellExtents%Zmin
            Cells(X, Y, Z)%Z_max = CellExtents%MyBase%Zmax
            Cells(X, Y, Z)%X_index = CellIndeces%X
            Cells(X, Y, Z)%Y_index = CellIndeces%Y
            Cells(X, Y, Z)%Z_index = CellIndeces%Z
            Cells(X, Y, Z)%Centroid = Centroid
            Cells(X, Y, Z)%CellType = CellType

            IF (PipeIndex .NE. -1) THEN
                Cells(X, Y, Z)%PipeIndex = PipeIndex
                CALL CartesianPipeCellInformation_ctor(Cells(X, Y, Z)%PipeCellData, Cells(X, Y, Z)%X_max
                     - Cells(X, Y, Z)%X_min, PipeCircuit%PipeSize, NumRadialCells, Depth(Cells(X, Y, Z)),
                      InsulationThickness, Mesh%Radial%RadialMeshThickness, Has%Insulation)
            END IF

        END DO !'z
    END DO !'y
END DO !'x

END SUBROUTINE

SUBROUTINE SetupCellNeighbors()

    INTEGER :: X, Y, Z
    REAL(r64) :: ThisCellCentroidX
    REAL(r64) :: ThisCellCentroidY
    REAL(r64) :: ThisCellCentroidZ

    REAL(r64) :: CellRightCentroidX
    REAL(r64) :: CellRightLeftWallX
    REAL(r64) :: CellLeftCentroidX
    REAL(r64) :: CellLeftRightWallX
    REAL(r64) :: LeftCellCentroidX
    REAL(r64) :: LeftCellRightWallX
    REAL(r64) :: RightCellCentroidX
    REAL(r64) :: RightCellLeftWallX

    REAL(r64) :: UpperCellCentroidY
    REAL(r64) :: UpperCellLowerWallY
    REAL(r64) :: LowerCellCentroidY
    REAL(r64) :: LowerCellUpperWallY

    REAL(r64) :: UpperZCellCentroidZ
    REAL(r64) :: UpperZCellLowerWallZ
    REAL(r64) :: LowerZCellCentroidZ
    REAL(r64) :: LowerZCellUpperWallZ

    DO Z = 0, UBOUND(Cells, 3)
        DO Y = 0, UBOUND(Cells, 2)
            DO X = 0, UBOUND(Cells, 1)

                !'for convenience
                ThisCellCentroidX = Cells(X, Y, Z)%Centroid%X
                ThisCellCentroidY = Cells(X, Y, Z)%Centroid%Y
                ThisCellCentroidZ = Cells(X, Y, Z)%Centroid%Z

                !'setup east/west cell neighbors
                IF (X == 0) THEN !'we have a left boundary, set east cell neighbor only
                    CellRightCentroidX = Cells(X + 1, Y, Z)%Centroid%X
                    CellRightLeftWallX = Cells(X + 1, Y, Z)%X_min
                    CALL AddNeighborInformation(X, Y, Z, Direction_PositiveX, CellRightCentroidX -
                         ThisCellCentroidX, CellRightLeftWallX - ThisCellCentroidX, CellRightCentroidX -
                         CellRightLeftWallX)
                    CALL AddNeighborInformation(X, Y, Z, Direction_NegativeX, 0.0d0, 0.0d0, 0.0d0)
                ELSE IF (X == UBOUND(Cells,1)) THEN !'we have a right bndy, set west cell neighbor only
                    CellLeftCentroidX = Cells(X - 1, Y, Z)%Centroid%X
                    CellLeftRightWallX = Cells(X - 1, Y, Z)%X_max
                    CALL AddNeighborInformation(X, Y, Z, Direction_NegativeX, ThisCellCentroidX -
                         CellLeftCentroidX, ThisCellCentroidX - CellLeftRightWallX, CellLeftRightWallX -
                         CellLeftCentroidX)
                    CALL AddNeighborInformation(X, Y, Z, Direction_PositiveX, 0.0d0, 0.0d0, 0.0d0)
                ELSE !'we have an internal node, set east/west cell neighbors
                    LeftCellCentroidX = Cells(X - 1, Y, Z)%Centroid%X
                    LeftCellRightWallX = Cells(X - 1, Y, Z)%X_max
                    RightCellCentroidX = Cells(X + 1, Y, Z)%Centroid%X
                    RightCellLeftWallX = Cells(X + 1, Y, Z)%X_min
                    CALL AddNeighborInformation(X, Y, Z, Direction_NegativeX, ThisCellCentroidX -
                         LeftCellCentroidX, ThisCellCentroidX - LeftCellRightWallX, LeftCellRightWallX -
                         LeftCellCentroidX)
                    CALL AddNeighborInformation(X, Y, Z, Direction_PositiveX, RightCellCentroidX -
                         ThisCellCentroidX, RightCellLeftWallX - ThisCellCentroidX, RightCellCentroidX -
                         RightCellLeftWallX)
                END IF

                !'setup north/south cell neighbors
                IF (Y == 0) THEN !'we have a lower boundary, set north cell neighbor only
                    UpperCellCentroidY = Cells(X, Y + 1, Z)%Centroid%Y
                    UpperCellLowerWallY = Cells(X, Y + 1, Z)%Y_min
                    CALL AddNeighborInformation(X, Y, Z, Direction_PositiveY, UpperCellCentroidY -
                         ThisCellCentroidY, UpperCellLowerWallY - ThisCellCentroidY, UpperCellCentroidY -
                         UpperCellLowerWallY)
                    CALL AddNeighborInformation(X, Y, Z, Direction_NegativeY, 0.0d0, 0.0d0, 0.0d0)
                ELSE IF (Y == UBOUND(Cells, 2)) THEN    !'we have an upper bndy, set lower cell neighbor only
                    LowerCellCentroidY = Cells(X, Y - 1, Z)%Centroid%Y
                    LowerCellUpperWallY = Cells(X, Y - 1, Z)%Y_max
                    CALL AddNeighborInformation(X, Y, Z, Direction_NegativeY, ThisCellCentroidY -
                         LowerCellCentroidY, ThisCellCentroidY - LowerCellUpperWallY, LowerCellUpperWallY -
                         LowerCellCentroidY)
                    CALL AddNeighborInformation(X, Y, Z, Direction_PositiveY, 0.0d0, 0.0d0, 0.0d0)
                ELSE !'we have an internal node, set north/south cell neighbors
                    UpperCellCentroidY = Cells(X, Y + 1, Z)%Centroid%Y
                    LowerCellCentroidY = Cells(X, Y - 1, Z)%Centroid%Y
                    UpperCellLowerWallY = Cells(X, Y + 1, Z)%Y_min
                    LowerCellUpperWallY = Cells(X, Y - 1, Z)%Y_max
                    CALL AddNeighborInformation(X, Y, Z, Direction_NegativeY, ThisCellCentroidY -
                         LowerCellCentroidY, ThisCellCentroidY - LowerCellUpperWallY, LowerCellUpperWallY -
                         LowerCellCentroidY)
                    CALL AddNeighborInformation(X, Y, Z, Direction_PositiveY, UpperCellCentroidY -
                         ThisCellCentroidY, UpperCellLowerWallY - ThisCellCentroidY, UpperCellCentroidY -
                         UpperCellLowerWallY)
                END IF

                !'setup forward/backward cell neighbors
                IF (Z==0) THEN !'we have a "lower" boundary, set forward cell neighbor only
                    UpperZCellCentroidZ = Cells(X, Y, Z + 1)%Centroid%Z
                    UpperZCellLowerWallZ = Cells(X, Y, Z + 1)%Z_min
                    CALL AddNeighborInformation(X, Y, Z, Direction_PositiveZ, UpperZCellCentroidZ -
                         ThisCellCentroidZ, UpperZCellLowerWallZ - ThisCellCentroidZ, UpperZCellCentroidZ -
                         UpperZCellLowerWallZ)
                    CALL AddNeighborInformation(X, Y, Z, Direction_NegativeZ, 0.0d0, 0.0d0, 0.0d0)
                ELSE IF (Z == UBOUND(Cells,3)) THEN !'we have an "upper" bndy, set "lower" cell neighbor only
                    LowerZCellCentroidZ = Cells(X, Y, Z - 1)%Centroid%Z
                    LowerZCellUpperWallZ = Cells(X, Y, Z - 1)%Z_max
                    CALL AddNeighborInformation(X, Y, Z, Direction_NegativeZ, ThisCellCentroidZ -
                         LowerZCellCentroidZ, ThisCellCentroidZ - LowerZCellUpperWallZ, LowerZCellUpperWallZ
                         - LowerZCellCentroidZ)
                    CALL AddNeighborInformation(X, Y, Z, Direction_PositiveZ, 0.0d0, 0.0d0, 0.0d0)
                ELSE
                    LowerZCellCentroidZ = Cells(X, Y, Z - 1)%Centroid%Z
                    UpperZCellCentroidZ = Cells(X, Y, Z + 1)%Centroid%Z
                    UpperZCellLowerWallZ = Cells(X, Y, Z + 1)%Z_min
                    LowerZCellUpperWallZ = Cells(X, Y, Z - 1)%Z_max
                    CALL AddNeighborInformation(X, Y, Z, Direction_NegativeZ, ThisCellCentroidZ -
                         LowerZCellCentroidZ, ThisCellCentroidZ - LowerZCellUpperWallZ, LowerZCellUpperWallZ
                         - LowerZCellCentroidZ)
                    CALL AddNeighborInformation(X, Y, Z, Direction_PositiveZ, UpperZCellCentroidZ -
                         ThisCellCentroidZ, UpperZCellLowerWallZ - ThisCellCentroidZ, UpperZCellCentroidZ -
                         UpperZCellLowerWallZ)
                END IF

            END DO
        END DO
    END DO

END SUBROUTINE

SUBROUTINE AddNeighborInformation(X, Y, Z, Direction, ThisCentroidToNeighborCentroid,
     ThisCentroidToNeighborWall, ThisWallToNeighborCentroid)

    INTEGER, INTENT(IN) :: X
    INTEGER, INTENT(IN) :: Y
    INTEGER, INTENT(IN) :: Z
    INTEGER, INTENT(IN) :: Direction !From Enum: Direction
    REAL(r64), INTENT(IN) :: ThisCentroidToNeighborCentroid
    REAL(r64), INTENT(IN) :: ThisCentroidToNeighborWall
    REAL(r64), INTENT(IN) :: ThisWallToNeighborCentroid

    TYPE(DirectionNeighbor_Dictionary), ALLOCATABLE, DIMENSION(:) :: PrevValues
    INTEGER :: PrevUbound
    IF (.NOT. ALLOCATED(Cells(X,Y,Z)%NeighborInformation)) THEN
        ALLOCATE(Cells(X,Y,Z)%NeighborInformation(0:0))
        PrevUBound = -1
    ELSE
        PrevUbound = UBOUND(Cells(X,Y,Z)%NeighborInformation, 1)
        ALLOCATE(PrevValues(0:PrevUbound))
        PrevValues = Cells(X,Y,Z)%NeighborInformation
        DEALLOCATE(Cells(X,Y,Z)%NeighborInformation)
        ALLOCATE(Cells(X,Y,Z)%NeighborInformation(0:PrevUbound+1))
        Cells(X,Y,Z)%NeighborInformation(0:PrevUbound) = PrevValues
    END IF

    Cells(X,Y,Z)%NeighborInformation(PrevUbound+1)%Direction = Direction
    Cells(X,Y,Z)%NeighborInformation(PrevUbound+1)%Value%ThisCentroidToNeighborCentroid =
         ThisCentroidToNeighborCentroid
    Cells(X,Y,Z)%NeighborInformation(PrevUbound+1)%Value%ThisCentroidToNeighborWall =
         ThisCentroidToNeighborWall
    Cells(X,Y,Z)%NeighborInformation(PrevUbound+1)%Value%ThisWallToNeighborCentroid =
         ThisWallToNeighborCentroid

    RETURN

END SUBROUTINE

SUBROUTINE SetupPipeCircuitInOutCells()

    LOGICAL ::  CircuitInletCellSet
    TYPE(CartesianCell) :: CircuitInletCell
    TYPE(CartesianCell) :: CircuitOutletCell
    TYPE(CartesianCell) :: SegmentInletCell
    TYPE(CartesianCell) :: SegmentOutletCell
    TYPE(PipeSegmentInfo) :: Segment
    INTEGER :: SegmentCtr

    CircuitInletCellSet = .FALSE.
    DO SegmentCtr = LBOUND(PipeCircuit%PipeSegments, 1), UBOUND(PipeCircuit%PipeSegments, 1)
        Segment = PipeCircuit%PipeSegments(SegmentCtr)
        SELECT CASE (Segment%FlowDirection)
            CASE (SegmentFlow_IncreasingZ)
                SegmentInletCell = Cells(segment%PipeCellCoordinates%X, segment%PipeCellCoordinates%Y, 0)
                SegmentOutletCell = Cells(segment%PipeCellCoordinates%X, segment%PipeCellCoordinates%Y,
                     UBOUND(Cells, 3))
            CASE (SegmentFlow_DecreasingZ)
                SegmentInletCell = Cells(segment%PipeCellCoordinates%X, segment%PipeCellCoordinates%Y, UBOUND
                     (Cells, 3))
                SegmentOutletCell = Cells(segment%PipeCellCoordinates%X, segment%PipeCellCoordinates%Y, 0)
        END SELECT
        IF (.NOT. CircuitInletCellSet) THEN
            CircuitInletCell = SegmentInletCell
            CircuitInletCellSet = .TRUE.
        END IF
        CircuitOutletCell = SegmentOutletCell
    END DO
    CALL PipeCircuitInfo_InitInOutCells(PipeCircuit, Cells(CircuitInletCell%X_index, CircuitInletCell%Y_index
         , CircuitInletCell%Z_index), Cells(CircuitOutletCell%X_index, CircuitOutletCell%Y_index,
         CircuitOutletCell%Z_index))

END SUBROUTINE

INTEGER FUNCTION GetCellWidthsCount(dir) RESULT(RetVal)

    INTEGER, INTENT(IN) :: dir !From Enum: RegionType

    SELECT CASE (dir)
    CASE (RegionType_XDirection)
        RetVal = Mesh%X%RegionMeshCount
    CASE (RegionType_YDirection)
        RetVal = Mesh%Y%RegionMeshCount
    CASE (RegionType_ZDirection)
        RetVal = Mesh%Z%RegionMeshCount
    END SELECT

    RETURN

END FUNCTION

SUBROUTINE GetCellWidths(g)

    TYPE(GridRegion), INTENT(IN OUT) :: g
    REAL(r64), ALLOCATABLE, DIMENSION(:) :: RetVal

    TYPE(DistributionStructure) :: ThisMesh
    REAL(r64) :: GridWidth
    INTEGER :: NumCellsOnEachSide
    REAL(r64) :: SummationTerm
    INTEGER :: I
    REAL(r64) :: CellWidth
    INTEGER :: SubIndex

    !'determine which mesh "direction" we are going to be using
    SELECT CASE (g%RegionType)
    CASE (RegionType_XDirection)
        ThisMesh = Mesh%X
        CASE (RegionType_YDirection)
            ThisMesh = Mesh%Y
        CASE (RegionType_ZDirection)
            ThisMesh = Mesh%Z
        CASE DEFAULT
            !Error
        END SELECT

        ALLOCATE(RetVal(0:ThisMesh%RegionMeshCount-1))

        GridWidth = g%Max - g%Min

        IF (ThisMesh%MeshDistribution == MeshDistribution_Uniform) THEN

            !we have it quite simple

            CellWidth = GridWidth / ThisMesh%RegionMeshCount

            DO I = 0, ThisMesh%RegionMeshCount - 1
                RetVal(I) = CellWidth
            END DO

        ELSEIF (ThisMesh%MeshDistribution == MeshDistribution_SymmetricGeometric) THEN

            !'then apply this "direction"'s conditions to generate a cell width array
            !'first get the total number of cells on this half of the region
            NumCellsOnEachSide = ThisMesh%RegionMeshCount / 2  !Already validated to be an even #

            !'calculate geometric series
            SummationTerm = 0.0d0
            DO I = 1, NumCellsOnEachSide
                SummationTerm = SummationTerm + ThisMesh%GeometricSeriesCoefficient ** (I - 1)
            END DO

            !'set up a list of cell widths for this region
            CellWidth = (GridWidth / 2) / SummationTerm
            RetVal(0) = CellWidth
            DO I = 1, NumCellsOnEachSide - 1
                CellWidth = CellWidth * ThisMesh%GeometricSeriesCoefficient
                RetVal(I) = CellWidth
            END DO
            SubIndex = NumCellsOnEachSide
            DO I = NumCellsOnEachSide-1, 0, -1
                SubIndex = SubIndex + 1
                RetVal(SubIndex) = RetVal(I)
            END DO

        END IF

        g%CellWidths = RetVal
        DEALLOCATE(RetVal)

    END SUBROUTINE

END MODULE

MODULE PiechowskiSimulationManager

    USE PiechowskiData
    USE Sim
    USE Extensions
    USE mCartesianCell
    IMPLICIT NONE
    PUBLIC

    !As much as I hate to do it, it is so so much easier to define them out here...
    INTEGER, ALLOCATABLE, DIMENSION(:) :: NeighborFieldCells
    INTEGER, ALLOCATABLE, DIMENSION(:) :: NeighborBoundaryCells

    !Pipe Circuit stuff
    REAL(r64), PARAMETER :: CurFluidDensity = 998.0d0
    REAL(r64), PARAMETER :: CurFluidViscosity = 0.0015d0
    REAL(r64), PARAMETER :: CurFluidConductivity = 0.58d0
    REAL(r64), PARAMETER :: CurFluidPrandtl = 7.0d0
    REAL(r64), PARAMETER :: CurFluidSpecificHeat = 4190.0d0
    REAL(r64), PARAMETER :: StagnantFluidConvCoeff = 200.0d0

    !Other?
    LOGICAL,   PARAMETER :: DoingFreezing = .FALSE.
    REAL(r64), PARAMETER :: BasementFloorConvCoeff = 23.0d0
    REAL(r64), PARAMETER :: BasementWallConvCoeff = 23.0d0
    REAL(r64), PARAMETER :: AirDensity = 1.22521d0 !'[kg/m3]
    REAL(r64), PARAMETER :: AirSpecificHeat = 1003d0 !'[J/kg-K]
    REAL(r64), PARAMETER :: RadiationAbsorption = 0.25d0 !0.333d0
    REAL(r64), PARAMETER :: SecondsPerHour = 3600.0d0
    REAL(r64), PARAMETER :: HoursPerDay = 24.0d0
    REAL(r64), PARAMETER :: NumSecondsPerDay = SecondsPerHour * HoursPerDay

    !Things that need to vary, even in the standalone simulation -- would come from heat pump sim
    REAL(r64) :: CurCircuitInletTemp = 23.0d0
    REAL(r64) :: CurCircuitFlowRate = 0.1321
    REAL(r64) :: CurCircuitHeatPumpQ = 500.0d0
    REAL(r64) :: BasementTemp = 21.0d0
    REAL(r64) :: CurSimTimeSeconds
    REAL(r64) :: CurSimTimeStepSize

    !Environmental conditions
    REAL(r64) :: CurAirTemp = 10.0d0 !-12.2d0 !Sim.CurSimConditions.CurTransient.DryBulb
    REAL(r64) :: CurWindSpeed = 2.6d0 !Sim.CurSimConditions.CurTransient.WindSpeed !'[m/s]
    REAL(r64) :: CurIncidentSolar = 0.0d0
    REAL(r64) :: CurRelativeHumidity = 100.0d0
    CONTAINS

    SUBROUTINE PerformSimulation(TimeStepIndex, TimeStepSize, NumIterationsUsed, ErrorsFound)

        LOGICAL, SAVE :: DoOneTimeInits = .TRUE.
        INTEGER, INTENT(IN) :: TimeStepIndex
        REAL(r64), INTENT(IN) :: TimeStepSize
        INTEGER, INTENT(INOUT) :: NumIterationsUsed
        LOGICAL, INTENT(INOUT) :: ErrorsFound
        LOGICAL :: hard_stop

        !Set the current sim time
        CurSimTimeStepSize = TimeStepSize
        CurSimTimeSeconds = TimeStepIndex * TimeStepSize

        !Do any one-time initializations
        IF (DoOneTimeInits) THEN
            CALL DoOneTimeInitializations()
            DoOneTimeInits = .FALSE.
        END IF

        ! allow the program to end cleanly if this file is found
        inquire(FILE=stopFile, EXIST=hard_stop)
        if (hard_stop) then
            stop 603
        end if

        !Update the temperature field
        CALL PerformSimulationLoop(NumIterationsUsed, ErrorsFound)

        !Do any post-processing
        !CALL PerformEnergyCalculations()

    END SUBROUTINE

    SUBROUTINE PerformSimulationLoop(NumIterationsUsed, ErrorsFound)

        LOGICAL :: NewTimeStep
        INTEGER :: IterationIndex
        LOGICAL :: FinishedIterationLoop
        LOGICAL, INTENT(INOUT) :: ErrorsFound
        INTEGER, INTENT(INOUT) :: NumIterationsUsed

        NewTimeStep = .TRUE. !TO be determined during simulation

        ! If we moved in time, we need to shift the previous values
        IF (NewTimeStep) THEN
            CALL ShiftTemperaturesForNewTimeStep()
        END IF

        ! Always do start of time step inits
        CALL DoStartOfTimeStepInitializations()

        ! Prepare the pipe circuit for calculations, but we'll actually do calcs at the iteration level
        IF (Has%PipeCircuit) CALL PreparePipeCircuitSimulation()

        ! Begin iterating for this time step
        DO IterationIndex = 1, SimControls%Cartesian%MaxIterationsPerTS

            !CALL DoStartOfIterationInitializations()

            CALL ShiftTemperaturesForNewIteration()

            IF (Has%PipeCircuit) CALL PerformPipeCircuitSimulation()

            CALL PerformTemperatureFieldUpdate()

        !     CALL PerformEnergyCalculations

            FinishedIterationLoop = .FALSE.
            ErrorsFound = .FALSE.
            CALL DoEndOfIterationOperations(IterationIndex, FinishedIterationLoop, ErrorsFound)
            IF (ErrorsFound) RETURN
            IF (FinishedIterationLoop) EXIT

        END DO

        NumIterationsUsed = MIN(IterationIndex, SimControls%Cartesian%MaxIterationsPerTS)

        RETURN

    END SUBROUTINE

    SUBROUTINE PerformTemperatureFieldUpdate()
```

```
    INTEGER :: X, Y, Z

    DO Z = 0, UBOUND(Cells, 3)
        DO Y = 0, UBOUND(Cells, 2)
            DO X = 0, UBOUND(Cells, 1)

                !'otherwise call the appropriate calculation routine
                SELECT CASE (Cells(X,Y,Z)%CellType)
                CASE (CellType_Pipe)
                    !'pipes are simulated separately
                CASE (CellType_GeneralField)
                    Cells(X,Y,Z)%MyBase%Temperature = EvaluateFieldCellTemperature(Cells(X,Y,Z))
                CASE (CellType_GroundSurface)
                    Cells(X,Y,Z)%MyBase%Temperature = EvaluateGroundSurfaceTemperature(Cells(X,Y,Z))
                CASE (CellType_FarfieldBoundary)
                    Cells(X,Y,Z)%MyBase%Temperature = EvaluateFarfieldBoundaryTemperature(Cells(X,Y,Z))
                CASE (CellType_BasementWall, CellType_BasementCorner, CellType_BasementFloor)
                    Cells(X,Y,Z)%MyBase%Temperature = EvaluateBasementCellTemperature(Cells(X,Y,Z))
                CASE (CellType_BasementCutAway)
                    Cells(X,Y,Z)%MyBase%Temperature = BasementTemp !To be updated during simulation
                CASE (CellType_AdiabaticWall)
                    Cells(X,Y,Z)%MyBase%Temperature = EvaluateAdiabaticSurfaceTemperature(Cells(X,Y,Z))
                END SELECT

            END DO
        END DO
    END DO

END SUBROUTINE

REAL(r64) FUNCTION EvaluateFieldCellTemperature(ThisCell) RESULT(RetVal)

    TYPE(CartesianCell), INTENT(IN) :: ThisCell

    REAL(r64) :: Numerator
    REAL(r64) :: Denominator
    REAL(r64) :: Beta
    REAL(r64) :: NeighborTemp
    REAL(r64) :: Resistance
    INTEGER :: DirectionCounter
    INTEGER :: CurDirection !From Enum: Direction

    !Set up once-per-cell items
    Numerator = 0.0
    Denominator = 0.0
    Beta = ThisCell%MyBase%Beta

    !add effect from cell history
    Numerator = Numerator + ThisCell%MyBase%Temperature_PrevTimeStep
    Denominator = Denominator + 1

    !determine the neighbor types based on cell location
    CALL EvaluateCellNeighborDirections(ThisCell)

    !loop across each direction in the simulation
    DO DirectionCounter = LBOUND(NeighborFieldCells,1), UBOUND(NeighborFieldCells,1)

        CurDirection = NeighborFieldCells(DirectionCounter)

        !'evaluate the transient expression terms
        CALL EvaluateNeighborCharacteristics(ThisCell, CurDirection, NeighborTemp, Resistance)
        Numerator = Numerator + (Beta / Resistance) * NeighborTemp
        Denominator = Denominator + Beta / Resistance

    END DO

    !'now that we have passed all directions, update the temperature
    RetVal = Numerator / Denominator

END FUNCTION

REAL(r64) FUNCTION EvaluateGroundSurfaceTemperature(cell) RESULT(RetVal)

    TYPE(CartesianCell), INTENT(IN) :: cell

    !declare some variables
    REAL(r64) :: Numerator
    REAL(r64) :: Denominator
    REAL(r64) :: Resistance
    REAL(r64) :: NeighborTemp
    REAL(r64) :: ThisNormalArea
    REAL(r64) :: IncidentHeatGain
    INTEGER :: DirectionCounter
    INTEGER :: CurDirection
    REAL(r64) :: AdiabaticMultiplier
    REAL(r64) :: Beta

    !evapotranspiration "inputs"
    REAL(r64), PARAMETER :: Latitude_Degrees = 36.010278d0
    REAL(r64), PARAMETER :: StMeridian_Degrees = 90.0d0    !Standard meridian, degrees W
    REAL(r64), PARAMETER :: Longitude_Degrees = 84.269722d0  !Longitude, degrees W
    REAL(r64), PARAMETER :: Elevation = 0.0d0 !units? sea level?
```

```
!evapotranspiration parameters
REAL(r64), PARAMETER :: MeanSolarConstant = 0.08196d0  ! 1367 [W/m2], entered in [MJ/m2-minute]
REAL(r64), PARAMETER :: A_s  = 0.25d0 !?
REAL(r64), PARAMETER :: B_s  = 0.5d0  !?
REAL(r64), PARAMETER :: Absor_Corrected = 0.77d0
REAL(r64), PARAMETER :: Convert_Wm2_To_MJhrmin = 3600.0d0 / 1000000.0d0
REAL(r64), PARAMETER :: Convert_MJhrmin_To_Wm2 = 1.0d0 / Convert_Wm2_To_MJhrmin
REAL(r64), PARAMETER :: Rho_water = 998.0d0  ![kg/m3]

!evapotranspiration calculated values
REAL(r64) :: Latitude_Radians
REAL(r64) :: DayOfYear
REAL(r64) :: HourOfDay
REAL(r64) :: CurSecondsIntoToday
REAL(r64) :: dr
REAL(r64) :: Declination
REAL(r64) :: b_sc
REAL(r64) :: Sc
REAL(r64) :: Hour_Angle
REAL(r64) :: X_sunset
REAL(r64) :: Sunset_Angle
REAL(r64) :: Altitude_Angle
REAL(r64) :: Solar_Angle_1
REAL(r64) :: Solar_Angle_2
REAL(r64) :: QRAD_A
REAL(r64) :: QRAD_SO
REAL(r64) :: Ratio_SO
REAL(r64), SAVE :: Last_Ratio_SO
REAL(r64) :: IncidentSolar_MJhrmin
REAL(r64) :: AbsorbedIncidentSolar_MJhrmin
REAL(r64) :: VaporPressureSaturated_kPa
REAL(r64) :: VaporPressureActual_kPa
REAL(r64) :: QRAD_NL
REAL(r64) :: NetIncidentRadiation_MJhr ![MJ/hr]
REAL(r64) :: NetIncidentRadiation_Wm2 ![W/m2]
REAL(r64) :: CN
REAL(r64) :: G_hr
REAL(r64) :: Cd
REAL(r64) :: Slope_S
REAL(r64) :: Pressure
REAL(r64) :: PsychrometricConstant
REAL(r64) :: EvapotransFluidLoss_mmhr
REAL(r64) :: EvapotransFluidLoss_mhr
REAL(r64) :: LatentHeatVaporization
REAL(r64) :: EvapotransHeatLoss_MJhrmin  ![MJ/m2-hr]
REAL(r64) :: EvapotransHeatLoss_Wm2      ![W/m2]
REAL(r64) :: CurAirTempK
LOGICAL, SAVE :: OneTimeOpenTempFile = .TRUE.

LOGICAL, PARAMETER :: DoingEvapotranspiration = .TRUE.

Numerator = 0.0
Denominator = 0.0
Resistance = 0.0
Beta = cell%MyBase%Beta
ThisNormalArea = NormalArea(cell, Direction_PositiveY)

!'add effect from previous time step
Numerator = Numerator + cell%MyBase%Temperature_PrevTimeStep
Denominator = Denominator + 1

!now that we aren't infinitesimal, we need to determine the neighbor types based on cell location
CALL EvaluateCellNeighborDirections(cell)

!loop over all regular neighbor cells, check if we have adiabatic on opposite surface
DO DirectionCounter = LBOUND(NeighborFieldCells,1), UBOUND(NeighborFieldCells,1)
    CurDirection = NeighborFieldCells(DirectionCounter)

    !If we have adiabatic z-faces, check if we are adjacent to one in the opposite direction
    !If we don't have adiabatic faces, we handle the boundary stuff below
    IF (Mesh%Z%BoundaryType == BoundaryType_Adiabatic) THEN
        IF ( (CurDirection==Direction_NegativeZ) .AND. (cell%Z_index==UBOUND(Cells,3)) ) THEN
            AdiabaticMultiplier = 2.0
        ELSE IF ( (CurDirection==Direction_PositiveZ) .AND. (cell%Z_index==0) ) THEN
            AdiabaticMultiplier = 2.0
        ELSE
            AdiabaticMultiplier = 1.0
        END IF
    ELSE
        AdiabaticMultiplier = 1.0
    END IF

    !Use the multiplier (either 1 or 2) to calculate the neighbor cell effects
    CALL EvaluateNeighborCharacteristics(cell, CurDirection, NeighborTemp, Resistance)
    Numerator = AdiabaticMultiplier * Numerator + (Beta / Resistance) * NeighborTemp
    Denominator = AdiabaticMultiplier * Denominator + (Beta / Resistance)

END DO

!do all non-adiabatic boundary types here
DO DirectionCounter = LBOUND(NeighborBoundaryCells,1), UBOUND(NeighborBoundaryCells,1)
    CurDirection = NeighborBoundaryCells(DirectionCounter)
```

```
        !x-direction will always be a farfield boundary
        !z-direction will be handled here if farfield, above if adiabatic
        !-y we don't handle here because -y will always be a neighbor cell, so handled above
        !+y will always be the outdoor air
        SELECT CASE (CurDirection)
            CASE (Direction_PositiveX, Direction_NegativeX)
                ! always farfield
                CALL EvaluateFarfieldCharacteristics(cell, CurDirection, NeighborTemp, Resistance)
                Numerator = Numerator + (Beta / Resistance) * NeighborTemp
                Denominator = Denominator + (Beta / Resistance)
            CASE (Direction_PositiveZ, Direction_NegativeZ)
                ! only if it is farfield
                IF (Mesh%Z%BoundaryType == BoundaryType_Farfield) THEN
                    CALL EvaluateFarfieldCharacteristics(cell, CurDirection, NeighborTemp, Resistance)
                    Numerator = Numerator + (Beta / Resistance) * NeighborTemp
                    Denominator = Denominator + (Beta / Resistance)
                END IF
            CASE (Direction_PositiveY)
                ! convection at the surface
                IF (CurWindSpeed .GT. 0.1) THEN
                    Resistance = 208 / (AirDensity * AirSpecificHeat * ThisNormalArea)
                    Numerator = Numerator + (Beta / Resistance) * CurAirTemp
                    Denominator = Denominator + (Beta / Resistance)
                ELSE
                    !Could include natural convection
                END IF
            CASE (Direction_NegativeY)
                !debug error, can't get here!
        END SELECT

    END DO

    ! Initialize, this variable is used for both evapotranspiration and non-ET cases, [W]
    IncidentHeatGain = 0.0d0

    ! Calculate any net heat gain into the cell from environment
    IF (DoingEvapotranspiration) THEN

        ! Latitude, converted to radians...positive for northern hemisphere, [radians]
        Latitude_Radians = Pi / 180.0d0 * Latitude_Degrees

        ! The day of year at this point in the simulation
        DayOfYear = INT( CurSimTimeSeconds / NumSecondsPerDay )

        ! The number of seconds into the current day
        CurSecondsIntoToday = INT( MOD( CurSimTimeSeconds, NumSecondsPerDay ) )

        ! The number of hours into today
        HourOfDay = INT( CurSecondsIntoToday / SecondsPerHour )

        ! For convenience convert to Kelvin once
        CurAirTempK = CurAirTemp + 273.15d0

        ! Calculate some angles
        dr   = 1.0d0 + 0.033d0 * COS(2.0d0 * Pi * DayOfYear / 365.0d0)
        Declination = 0.409d0 * SIN(2.0d0 * Pi / 365.0d0 * DayOfYear - 1.39d0)
        b_SC = 2.0d0 * Pi * (DayOfYear - 81.0d0)/364.0d0
        Sc   = 0.1645d0 * SIN(2.0d0 * b_SC) - 0.1255d0 * COS(b_SC) - 0.025d0 * SIN(b_SC)
        Hour_Angle = Pi / 12.0d0 * ( ( (HourOfDay - 0.5d0) + 0.06667d0 * (StMeridian_Degrees -
             Longitude_Degrees) + Sc) - 12.0d0)

        !For HOUR_INADAY-0.5 not HOUR_INADAY+0.5, as HOUR_INADAY from 0 to 1, it shows 1 not zero here.
        !Lz longitude of the centre of the local time zone [degrees west of Greenwich].
        ! For example, Lz = 75, 90, 105 and 120     for the Eastern, Central, Rocky Mountain and Pacific time
              zones (United States)
        !and Lz = 0     for Greenwich, 330     for Cairo (Egypt), and 255     for Bangkok (Thailand),

        ! Calculate sunset something, and constrain to a minimum of 0.000001
        X_sunset = 1.0d0 - TAN(Latitude_Radians)**2.0d0 * TAN(Declination)**2.0d0
        X_sunset = MAX(X_sunset, 0.000001d0)

        ! Find sunset angle
        Sunset_angle = Pi / 2.0d0 - ATAN(-TAN(Latitude_Radians) * TAN(Declination) / X_sunset**0.5d0 )

        ! Find the current sun angle
        Altitude_Angle = ASIN( SIN(Latitude_Radians) * SIN(Declination) + COS(Latitude_Radians) * COS(
             Declination) * COS(Hour_Angle) )

        ! Find solar angles
        Solar_angle_1 = Hour_Angle - Pi / 24.0d0
        Solar_angle_2 = Hour_Angle + Pi / 24.0d0

        ! Constrain solar angles
        IF(Solar_angle_1 .LT. -Sunset_angle ) Solar_angle_1 = -Sunset_angle
        IF(Solar_angle_2 .LT. -Sunset_angle ) Solar_angle_2 = -Sunset_angle
        IF(Solar_angle_1 .GT.  Sunset_angle ) Solar_angle_1 =  Sunset_angle
        IF(Solar_angle_2 .GT.  Sunset_angle ) Solar_angle_2 =  Sunset_angle
        IF(Solar_angle_1 .GT.  Solar_angle_2) Solar_angle_1 =  Solar_angle_2

        ! Convert input solar radiation [w/m2] into units for ET model, [MJ/hr-min]
        IncidentSolar_MJhrmin = CurIncidentSolar * Convert_Wm2_To_MJhrmin
        ! Calculate another Q term...
        QRAD_a = 12.0d0 * 60.0d0 / Pi * MeanSolarConstant * dr * &
            ( (Solar_angle_2 - Solar_angle_1) * SIN(Latitude_Radians) * SIN(Declination) + COS(
                 Latitude_Radians) * COS(Declination) * (SIN(Solar_angle_2)-SIN(Solar_angle_1)))

        ! Calculate another Q term...
        QRAD_SO = ( A_s + B_s + 0.00002d0 * Elevation ) * QRAD_a

        ! Correct the Qrad term ... better way??
        IF (IncidentSolar_MJhrmin .LT. 0.01d0) THEN
            Ratio_SO = LAST_RATIO_SO
        ELSE
            IF (QRAD_SO /= 0.d0) THEN
                Ratio_SO = IncidentSolar_MJhrmin / QRAD_SO
            ELSE
                Ratio_SO = 1.0d0
            END IF
        END IF

        ! Constrain Ratio_SO
        Ratio_SO = MIN(Ratio_SO, 1.0d0)
        Ratio_SO = MAX(Ratio_SO, 0.3d0)

        ! Store the previous
        LAST_RATIO_SO=Ratio_SO

        ! Calculate another Q term, [MJ/hr-min]
        AbsorbedIncidentSolar_MJhrmin = ABSOR_CORRECTED * IncidentSolar_MJhrmin

        ! Calculate saturated vapor pressure, [kPa]
        VaporPressureSaturated_kPa = 0.6108d0 * EXP(17.27d0 * CurAirTemp / (CurAirTemp + 237.3d0))

        ! Calculate actual vapor pressure, [kPa]
        VaporPressureActual_kPa = VaporPressureSaturated_kPa * CurRelativeHumidity / 100.0d0

        ! Calculate another Q term, [MJ/m2-hr]
        QRAD_NL = 2.042D-10 * CurAirTempK**4.0d0 * (0.34d0 - 0.14d0 * SQRT(VaporPressureActual_kPa)) * (1.35
             d0 * Ratio_SO - 0.35d0)

        ! Calculate another Q term, [MJ/hr]
        NetIncidentRadiation_MJhr = AbsorbedIncidentSolar_MJhrmin - QRAD_NL

        ! ?
        Cn = 37.0d0

        ! Check whether there was sun
        IF (NetIncidentRadiation_MJhr .LT. 0.0)THEN
            G_hr = 0.5d0 * NetIncidentRadiation_MJhr
            Cd = 0.96d0
        ELSE
            G_hr = 0.1d0 * NetIncidentRadiation_MJhr
            Cd = 0.24d0
        END IF

        ! Just For Check
        ! Lu Xing Sep 22 2009

        Slope_S = 2503.0d0 * EXP(17.27d0 * CurAirTemp / (CurAirTemp + 237.3d0)) / (CurAirTemp+237.3d0)**2
        Pressure = 98.0d0
        PsychrometricConstant = 0.665E-3 * Pressure

        ! Evapotranspiration constant, [mm/hr]
        EvapotransFluidLoss_mmhr = (0.408d0 * Slope_s * (NetIncidentRadiation_MJhr - G_hr) +
             PsychrometricConstant * (Cn / CurAirTempK) * Curwindspeed * (VaporPressureSaturated_kPa -
             VaporPressureActual_kPa)) &
                 / (Slope_s + PsychrometricConstant * (1 + Cd * CurWindSpeed))

        ! Convert units, [m/hr]
        EvapotransFluidLoss_mhr = EvapotransFluidLoss_mmhr / 1000.0d0

        ! Calculate latent heat, [MJ/kg]
        ! Full formulation is cubic: L(T) =        0 .0000614342 * T**3 + 0.00158927 * T**2          2.36418
             * T + 2500.79[5]
        ! In: Cubic fit to Table 2.1,p.16, Textbook: R.R.Rogers & M.K. Yau, A Short Course in Cloud Physics,
             3e,(1989), Pergamon press
        ! But a linear relation should suffice; -for now using the previous time step temperature as an
             approximation to help ensure stability
        LatentHeatVaporization = 2.501d0 - 2.361d-3 * cell%MyBase%Temperature_PrevTimeStep

        ! Calculate evapotranspiration heat loss, [MJ/m2-hr]
        EvapotransHeatLoss_MJhrmin = RHO_water * EvapotransFluidLoss_mhr * LatentHeatVaporization

        ! Convert net incident solar units, [W/m2]
        NetIncidentRadiation_Wm2 = NetIncidentRadiation_MJhr * Convert_MJhrmin_To_Wm2

        ! Convert evapotranspiration units, [W/m2]
        EvapotransHeatLoss_Wm2 = EvapotransHeatLoss_MJhrmin * Convert_MJhrmin_To_Wm2

        ! Calculate overall net heat ?gain? into the cell, [W]
        IncidentHeatGain = (NetIncidentRadiation_Wm2 - EvapotransHeatLoss_Wm2) * ThisNormalArea

    ELSE
```

```
        !'calculate solar radiation into the cell using simple radiation and no evapotranspiration
        IncidentHeatGain = CurIncidentSolar * RadiationAbsorption * ThisNormalArea

    END IF

    ! Add any solar/evapotranspiration heat gain here
    Numerator = Numerator + Beta * IncidentHeatGain

    ! Calculate the return temperature and leave
    RetVal = Numerator / Denominator

    RETURN

END FUNCTION

REAL(r64) FUNCTION EvaluateAdiabaticSurfaceTemperature(cell) RESULT(RetVal)

    TYPE(CartesianCell), INTENT(IN) :: cell

    REAL(r64) :: Numerator
    REAL(r64) :: Denominator
    REAL(r64) :: Resistance
    REAL(r64) :: NeighborTemp
    REAL(r64) :: Beta
    INTEGER :: DirectionCounter
    INTEGER :: CurDirection
    REAL(r64) :: AdiabaticMultiplier

    Numerator = 0.0
    Denominator = 0.0
    Resistance = 0.0
    Beta = cell%MyBase%Beta

    !'add effect from previous time step
    Numerator = Numerator + cell%MyBase%Temperature_PrevTimeStep
    Denominator = Denominator + 1

    !now that we aren't infinitesimal, we need to determine the neighbor types based on cell location
    CALL EvaluateCellNeighborDirections(cell)

    DO DirectionCounter = LBOUND(NeighborFieldCells,1), UBOUND(NeighborFieldCells,1)
        CurDirection = NeighborFieldCells(DirectionCounter)
        AdiabaticMultiplier = 1.0

        ! There are only a few cases for adiabatic cells to be handled here
        ! These cases must be validated during mesh development as they aren't here
        ! For example, the +x case below will only be hit if the celltype is actually
        !   assigned to be Adiabatic...which only happens if the mesh dev engine
        !   recognizes that there is in fact a basement, and the boundary type is
        !   specified as adiabatic.
        SELECT CASE (CurDirection)
            CASE (Direction_PositiveZ) ! Case: front face looking in +z direction
                IF (cell%Z_index == 0) AdiabaticMultiplier = 2.0
            CASE (Direction_NegativeZ) ! Case: back face looking in -z direction
                IF (cell%Z_index == UBOUND(Cells,3)) AdiabaticMultiplier = 2.0
            CASE (Direction_PositiveX) ! Case: Under basement floor, far left cell
                IF (cell%X_index == 0) AdiabaticMultiplier = 2.0
            CASE (Direction_NegativeY) ! Case: basement wall ground surface boundary
                !Not sure if this is ever hit (it should be a basement wall celltype)
                IF (cell%Y_index == UBOUND(Cells,2)) AdiabaticMultiplier = 2.0
        END SELECT

        !Use the multiplier (either 1 or 2) to calculate the neighbor cell effects
        CALL EvaluateNeighborCharacteristics(cell, CurDirection, NeighborTemp, Resistance)
        Numerator = AdiabaticMultiplier * Numerator + (Beta / Resistance) * NeighborTemp
        Denominator = AdiabaticMultiplier * Denominator + (Beta / Resistance)

    END DO

    RetVal = Numerator / Denominator

    RETURN

END FUNCTION

REAL(r64) FUNCTION EvaluateBasementCellTemperature(cell) RESULT(RetVal)

    TYPE(CartesianCell), INTENT(IN) :: cell

    REAL(r64) :: Numerator
    REAL(r64) :: Denominator
    REAL(r64) :: Beta
    REAL(r64) :: Resistance
    REAL(r64) :: NeighborTemp
    INTEGER :: DirectionCounter
    INTEGER :: CurDirection
    REAL(r64) :: AdiabaticMultiplier

    Numerator = 0.0
    Denominator = 0.0
    Resistance = 0.0
    Beta = cell%MyBase%Beta
    !add effect from previous time step
    Numerator = Numerator + cell%MyBase%Temperature_PrevTimeStep
    Denominator = Denominator + 1

    !now that we aren't infinitesimal, we need to determine the neighbor types based on cell location
    CALL EvaluateCellNeighborDirections(cell)

    DO DirectionCounter = LBOUND(NeighborFieldCells,1), UBOUND(NeighborFieldCells,1)
        CurDirection = NeighborFieldCells(DirectionCounter)
        AdiabaticMultiplier = 1.0

        ! The basement cells line up a lot with the adiabatic cell routine
        ! These cases must be validated during mesh development as they aren't here
        ! For example, the +x case below will only be hit if the celltype is actually
        !   assigned to be Adiabatic...which only happens if the mesh dev engine
        !   recognizes that there is in fact a basement, and the boundary type is
        !   specified as adiabatic.
        SELECT CASE (CurDirection)
            CASE (Direction_PositiveZ) ! Case: front face looking in +z direction
                IF (cell%Z_index == 0) AdiabaticMultiplier = 2.0
            CASE (Direction_NegativeZ) ! Case: back face looking in -z direction
                IF (cell%Z_index == UBOUND(Cells,3)) AdiabaticMultiplier = 2.0
            CASE (Direction_PositiveX) ! Case: Basement floor, far left cell
                IF (cell%X_index == 0) AdiabaticMultiplier = 2.0
            CASE (Direction_NegativeY) ! Case: basement wall ground surface boundary
                IF (cell%Y_index == UBOUND(Cells,2)) AdiabaticMultiplier = 2.0
        END SELECT

        !Use the multiplier (either 1 or 2) to calculate the neighbor cell effects
        CALL EvaluateNeighborCharacteristics(cell, CurDirection, NeighborTemp, Resistance)
        Numerator = AdiabaticMultiplier * Numerator + (Beta / Resistance) * NeighborTemp
        Denominator = AdiabaticMultiplier * Denominator + (Beta / Resistance)

    END DO

    RetVal = Numerator / Denominator

    RETURN

END FUNCTION

REAL(r64) FUNCTION EvaluateFarfieldBoundaryTemperature(cell) RESULT(RetVal)

    TYPE(CartesianCell), INTENT(IN) :: cell

    REAL(r64) :: Numerator
    REAL(r64) :: Denominator
    REAL(r64) :: Beta
    REAL(r64) :: Resistance
    INTEGER :: DirectionCounter
    INTEGER :: CurDirection
    REAL(r64) :: NeighborTemp

    Numerator = 0.0
    Denominator = 0.0
    Resistance = 0.0
    Beta = cell%MyBase%Beta

    !add effect from previous time step
    Numerator = Numerator + cell%MyBase%Temperature_PrevTimeStep
    Denominator = Denominator + 1

    !now that we aren't infinitesimal, we need to determine the neighbor types based on cell location
    CALL EvaluateCellNeighborDirections(cell)

    !This may be incomplete, as there may need to be adiabatic conditions to be handled here as well

    !Do all neighbor cells
    DO DirectionCounter = LBOUND(NeighborFieldCells,1), UBOUND(NeighborFieldCells,1)
        CurDirection = NeighborFieldCells(DirectionCounter)
        CALL EvaluateNeighborCharacteristics(cell, CurDirection, NeighborTemp, Resistance)
        Numerator = Numerator + (Beta / Resistance) * NeighborTemp
        Denominator = Denominator + (Beta / Resistance)
    END DO

    !Then all farfield boundaries
    DO DirectionCounter = LBOUND(NeighborBoundaryCells,1), UBOUND(NeighborBoundaryCells,1)
        CurDirection = NeighborBoundaryCells(DirectionCounter)
        CALL EvaluateFarfieldCharacteristics(cell, CurDirection, NeighborTemp, Resistance)
        Numerator = Numerator + (Beta / Resistance) * NeighborTemp
        Denominator = Denominator + (Beta / Resistance)
    END DO

    RetVal = Numerator / Denominator

    RETURN

END FUNCTION

SUBROUTINE EvaluateFarfieldCharacteristics(cell, direction, neighbortemp, resistance)

    TYPE(CartesianCell), INTENT(IN) :: cell
    INTEGER, INTENT(IN) :: direction
    REAL(r64), INTENT(OUT) :: neighbortemp
    REAL(r64), INTENT(OUT) :: resistance

    REAL(r64) :: distance

    SELECT CASE (direction)
    CASE(Direction_NegativeX, Direction_PositiveX)
        distance = (width(cell) / 2.0)
    CASE(Direction_NegativeY, Direction_PositiveY)
        distance = (height(cell) / 2.0)
    CASE(Direction_NegativeZ, Direction_PositiveZ)
        distance = (depth(cell) / 2.0)
    END SELECT

    !previously this distance was *again* divided by 2, a bug I believe, removed for now...
    resistance = distance / (cell%mybase%properties%conductivity * NormalArea(cell, direction))
    neighbortemp = GetFarfieldTemp(cell)

    RETURN

END SUBROUTINE

REAL(r64) FUNCTION GetFarfieldTemp(cell) RESULT(RetVal)

    TYPE(CartesianCell), INTENT(IN) :: cell

    REAL(r64) :: ScaledDepth
    REAL(r64) :: z
    REAL(r64) :: Term1
    REAL(r64) :: Term2

    SELECT CASE(Farfield%Model)
    CASE(FarfieldModel_Constant)
        RetVal = Farfield%Constant%Temperature
    CASE(FarfieldModel_ConstantLinear)
        ScaledDepth = (Extents%Ymax - cell%Centroid%Y) / Extents%Ymax
        RetVal = Farfield%ConstantLinear%SurfaceTemperature + Farfield%ConstantLinear%Slope * ScaledDepth
    CASE(FarfieldModel_KusudaAchenbach)
        z = Extents%Ymax - cell%Centroid%Y
        Term1 = -z * SQRT(PI / (SecondsInYear * BaseThermalPropertySet_Diffusivity(GroundProperties)))
        Term2 = (2 * PI / SecondsInYear) * (CurSimTimeSeconds - Farfield%KusudaAchenbach%
             PhaseShiftOfMinGroundTemp - (z / 2) * SQRT(SecondsInYear / (PI *
             BaseThermalPropertySet_Diffusivity(GroundProperties))))
        RetVal = Farfield%KusudaAchenbach%AverageGroundTemperature - Farfield%KusudaAchenbach%
             AverageGroundTemperatureAmplitude * EXP(Term1) * COS(Term2)
    END SELECT

END FUNCTION

SUBROUTINE PreparePipeCircuitSimulation()

    !In standalone, just pass the fluid through, no heat pump
    !In EnergyPlus, this will be where we initialize the entering conditions

    TYPE(CartesianCell) :: CellToCheck
    TYPE(RadialCellInformation) :: PipeCell
    TYPE(FluidCellInformation) :: FluidCell

    REAL(r64) :: Density
    REAL(r64) :: Viscosity
    REAL(r64) :: Conductivity
    REAL(r64) :: Prandtl
    REAL(r64) :: Area_c
    REAL(r64) :: Velocity
    REAL(r64) :: ConvCoefficient
    REAL(r64) :: Reynolds
    REAL(r64) :: ExponentTerm
    REAL(r64) :: Nusselt
    REAL(r64) :: CurCircuitOutletTemp
    REAL(r64) :: SpecificHeat

    !Setup circuit flow conditions -- convection coefficient
    CellToCheck = PipeCircuit%CircuitInletCell
    PipeCell = CellToCheck%PipeCellData%Pipe
    FluidCell = CellToCheck%PipeCellData%Fluid
    Density = PipeCircuit%CurFluidPropertySet%MyBase%Density
    Viscosity = PipeCircuit%CurFluidPropertySet%Viscosity
    Conductivity = PipeCircuit%CurFluidPropertySet%MyBase%Conductivity
    Prandtl = PipeCircuit%CurFluidPropertySet%Prandtl
    SpecificHeat = PipeCircuit%CurFluidPropertySet%MyBase%SpecificHeat
    Area_c = (Pi/4.0) * PipeCircuit%PipeSize%InnerDia**2
    Velocity = CurCircuitFlowRate / (Density * Area_c)
    IF (Velocity > 0) THEN
        Reynolds = Density * PipeCircuit%PipeSize%InnerDia * Velocity / Viscosity
        IF ( FluidCell%MyBase%Temperature > PipeCell%MyBase%Temperature ) THEN
            ExponentTerm = 0.3
        ELSE
            ExponentTerm = 0.4
        END IF
        Nusselt = 0.023 * Reynolds**(4./5.) * Prandtl**ExponentTerm
        ConvCoefficient = Nusselt * Conductivity / PipeCircuit%PipeSize%InnerDia
    ELSE
        ConvCoefficient = StagnantFluidConvCoeff
    END IF

    PipeCircuit%CurCircuitConvectionCoefficient = ConvCoefficient

    !Setup circuit entering conditions here -- heat pump model
    CurCircuitOutletTemp = PipeCircuit%CircuitOutletCell%PipeCellData%Fluid%MyBase%Temperature

    CurCircuitInletTemp = CurCircuitOutletTemp + (CurCircuitHeatPumpQ / (CurCircuitFlowRate * SpecificHeat))

    RETURN

END SUBROUTINE

SUBROUTINE PerformPipeCircuitSimulation()

    TYPE(CartesianCell) :: UpstreamCell
    REAL(r64) :: CircuitCrossTemp
    REAL(r64) :: FlowRate
    REAL(r64) :: EnteringTemp
    INTEGER :: SegmentCtr
    INTEGER :: SegmentCellCtr
    TYPE(PipeSegmentInfo) :: Segment
    INTEGER :: StartingZ
    INTEGER :: EndingZ
    INTEGER :: Increment
    INTEGER :: PipeX
    INTEGER :: PipeY
    INTEGER :: Zindex

    !retrieve initial conditions from the data structure
    !these have been set either by the init routine or by the heat pump routine
    FlowRate = CurCircuitFlowRate !PipeCircuit.CurCircuitFlowRate
    EnteringTemp = CurCircuitInletTemp !PipeCircuit.CircuitInletCell.PipeCellData.Fluid.Temperature

    !initialize
    SegmentCellCtr = 0

    !'loop across all segments (pipes) of the circuit
    DO SegmentCtr = LBOUND(PipeCircuit%PipeSegments,1), UBOUND(PipeCircuit%PipeSegments,1)

        Segment = PipeCircuit%PipeSegments(SegmentCtr)

        !'set simulation flow direction
        SELECT CASE (Segment%FlowDirection)
        CASE (SegmentFlow_IncreasingZ)
            StartingZ = 0
            EndingZ = UBOUND(Cells, 3)
            Increment = 1
        CASE (SegmentFlow_DecreasingZ)
            StartingZ = UBOUND(Cells, 3)
            EndingZ = 0
            Increment = -1
        END SELECT

        !'find the cell we are working on in order to retrieve cell and neighbor information
        PipeX = Segment%PipeCellCoordinates%X
        PipeY = Segment%PipeCellCoordinates%Y

        !'loop across all z-direction indeces
        DO Zindex = StartingZ, EndingZ, Increment

            !'overall cell segment counter
            SegmentCellCtr = SegmentCellCtr + 1

            IF (SegmentCellCtr == 1) THEN
                !'we have the very first cell, need to pass in circuiting entering temperature
                CALL PerformPipeCellSimulation(Cells(PipeX, PipeY, Zindex), FlowRate, EnteringTemp)
            ELSE
                !'we don't have the first cell so just normal simulation
                IF (Zindex == EndingZ) THEN
                    !get entering conditions from upstream cell
                    UpstreamCell = Cells(PipeX, PipeY, Zindex - Increment)
                    !simulate current cell using upstream as entering conditions
                    CALL PerformPipeCellSimulation(Cells(PipeX, PipeY, Zindex), FlowRate, UpstreamCell%
                         PipeCellData%Fluid%MyBase%Temperature)
                    !store this outlet condition to be passed to the next segment
                    CircuitCrossTemp = Cells(PipeX, PipeY, Zindex)%PipeCellData%Fluid%MyBase%Temperature
                ELSE IF (Zindex == StartingZ) THEN
                    !we are starting another segment, use the previous cross temperature
                    CALL PerformPipeCellSimulation(Cells(PipeX, PipeY, Zindex), FlowRate, CircuitCrossTemp)
                ELSE
                    !we are in an interior node, so just get the upstream cell and use the main simulation
                    UpstreamCell = Cells(PipeX, PipeY, Zindex - Increment)
                    CALL PerformPipeCellSimulation(Cells(PipeX, PipeY, Zindex), FlowRate, UpstreamCell%
                         PipeCellData%Fluid%MyBase%Temperature)
                END IF
            END IF

        END DO

    END DO

    RETURN
```

```
END SUBROUTINE

SUBROUTINE PerformPipeCellSimulation(ThisCell, FlowRate, EnteringTemp)

    TYPE(CartesianCell), INTENT(IN OUT) :: ThisCell
    REAL(r64), INTENT(IN) :: FlowRate
    REAL(r64), INTENT(IN) :: EnteringTemp

    INTEGER :: Iter
    REAL(r64) :: MaxDeviationAmount

    DO Iter = 1, SimControls%Radial%MaxIterationsPerTS

        !'shift all the pipe related temperatures for the next internal pipe iteration
        CALL ShiftPipeTemperaturesForNewIteration(ThisCell)

        !'simulate the funny interface soil cell between the radial and cartesian systems
        CALL SimulateRadialToCartesianInterface(ThisCell)

        !'simulate the outermost radial slice
        CALL SimulateOuterMostRadialSoilSlice(ThisCell)

        !'we only need to simulate these if they actually exist!
        IF (SIZE(ThisCell%PipeCellData%Soil) > 1) THEN

            !'simulate all interior radial slices
            CALL SimulateAllInteriorRadialSoilSlices(ThisCell)

            !'simulate the innermost radial soil slice
            CALL SimulateInnerMostRadialSoilSlice(ThisCell)

        END IF

        IF (Has%Insulation) THEN
            CALL SimulateRadialInsulationCell(ThisCell)
        END IF

        !'simulate the pipe cell
        CALL SimulateRadialPipeCell(ThisCell, FlowRate, PipeCircuit%CurCircuitConvectionCoefficient)

        !'simulate the water cell since there appears to be flow
        CALL SimulateFluidCell(ThisCell, FlowRate, PipeCircuit%CurCircuitConvectionCoefficient, EnteringTemp)

        !'check convergence
        IF (IsConverged_PipeCurrentToPrevIteration(ThisCell, MaxDeviationAmount)) EXIT

    END DO

    RETURN

END SUBROUTINE

SUBROUTINE SimulateRadialToCartesianInterface(ThisCell)

    TYPE(CartesianCell), INTENT(IN OUT) :: ThisCell

    !'placeholder variables
    REAL(r64) :: Numerator
    REAL(r64) :: Denominator
    REAL(r64) :: Resistance
    REAL(r64) :: Beta
    INTEGER :: DirCtr
    INTEGER :: Dir
    REAL(r64) :: NeighborTemp

    TYPE(RadialCellInformation) :: OutermostRadialCell

    INTEGER, DIMENSION(4), PARAMETER :: Directions = (/Direction_NegativeX, Direction_NegativeY,
         Direction_PositiveX, Direction_PositiveY/)

    Numerator = 0.0d0
    Denominator = 0.0d0

    !'convenience variables
    OutermostRadialCell = ThisCell%PipeCellData%Soil(UBOUND(ThisCell%PipeCellData%Soil,1))

    !'retrieve beta
    Beta = ThisCell%MyBase%Beta

    !'add effects from this cell history
    Numerator = Numerator + ThisCell%MyBase%Temperature_PrevTimeStep
    Denominator = Denominator + 1

    !'add effects from outermost radial cell
    Resistance = LOG(OutermostRadialCell%OuterRadius / OutermostRadialCell%RadialCentroid) / (2 * Pi * Depth(
         ThisCell) * ThisCell%MyBase%Properties%Conductivity)
    Numerator = Numerator + (Beta / Resistance) * OutermostRadialCell%MyBase%Temperature
    Denominator = Denominator + (Beta / Resistance)

    !'add effects from neighboring Cartesian cells
    DO DirCtr = LBOUND(Directions,1), UBOUND(Directions,1)
        Dir = Directions(DirCtr)
        !'get info about cartesian neighbors
        CALL EvaluateNeighborCharacteristics(ThisCell, Dir, NeighborTemp, Resistance)

        !'add to the numerator and denominator expressions
        Numerator = Numerator + (Beta / Resistance) * NeighborTemp
        Denominator = Denominator + (Beta / Resistance)

    END DO

    !'calculate the new temperature
    ThisCell%MyBase%Temperature = Numerator / Denominator

    RETURN

END SUBROUTINE

SUBROUTINE SimulateOuterMostRadialSoilSlice(ThisCell)

    TYPE(CartesianCell), INTENT(IN OUT) :: ThisCell

    !'placeholder variables
    REAL(r64) :: Numerator
    REAL(r64) :: Denominator
    REAL(r64) :: Resistance
    REAL(r64) :: Beta

    INTEGER :: MaxRadialIndex
    TYPE(RadialCellInformation) :: ThisRadialCell
    TYPE(RadialCellInformation) :: NextOuterRadialCell

    Numerator = 0.0d0
    Denominator = 0.0d0
    Resistance = 0.0d0

    !'convenience variables
    MaxRadialIndex = UBOUND(ThisCell%PipeCellData%Soil,1)
    ThisRadialCell = ThisCell%PipeCellData%Soil(MaxRadialIndex)
    IF (SIZE(ThisCell%PipeCellData%Soil)==1) THEN
        IF (Has%Insulation) THEN
            NextOuterRadialCell = ThisCell%PipeCellData%Insulation
        ELSE
            NextOuterRadialCell = ThisCell%PipeCellData%Pipe
        END IF
    ELSE
        NextOuterRadialCell = ThisCell%PipeCellData%Soil(MaxRadialIndex - 1)
    END IF

    !'any broadly defined variables
    Beta = ThisRadialCell%MyBase%Beta

    !'add effects from this cell history
    Numerator = Numerator + ThisRadialCell%MyBase%Temperature_PrevTimeStep
    Denominator = Denominator + 1

    !'add effects from interface cell
    Resistance = LOG(ThisRadialCell%OuterRadius / ThisRadialCell%RadialCentroid) / &
                    (2 * Pi * Depth(ThisCell) * ThisRadialCell%MyBase%Properties%Conductivity)
    Numerator = Numerator + (Beta / Resistance) * ThisCell%MyBase%Temperature
    Denominator = Denominator + (Beta / Resistance)

    !'add effects from inner radial cell
    Resistance = (LOG(ThisRadialCell%RadialCentroid / ThisRadialCell%InnerRadius) / &
                    (2 * Pi * Depth(ThisCell) * ThisRadialCell%MyBase%Properties%Conductivity)) &
                + (LOG(NextOuterRadialCell%OuterRadius/NextOuterRadialCell%RadialCentroid) / &
                    (2 * Pi * Depth(ThisCell) * NextOuterRadialCell%MyBase%Properties%Conductivity))
    Numerator = Numerator + (Beta / Resistance) * NextOuterRadialCell%MyBase%Temperature
    Denominator = Denominator + (Beta / Resistance)

    !'calculate the new temperature
    ThisCell%PipeCellData%Soil(MaxRadialIndex)%MyBase%Temperature = Numerator / Denominator

END SUBROUTINE

SUBROUTINE SimulateAllInteriorRadialSoilSlices(ThisCell)

    TYPE(CartesianCell), INTENT(IN OUT) :: ThisCell

    !'placeholder variables
    REAL(r64) :: Numerator
    REAL(r64) :: Denominator
    REAL(r64) :: Resistance
    REAL(r64) :: Beta

    INTEGER :: MaxRadialIndex
    TYPE(RadialCellInformation) :: ThisRadialCell
    TYPE(RadialCellInformation) :: InnerRadialCell
    TYPE(RadialCellInformation) :: oUTerRadialCell
    INTEGER :: rCtr

    Numerator = 0.0d0
    Denominator = 0.0d0
    DO rCtr = UBOUND(ThisCell%PipeCellData%Soil,1)-1, 1, -1

        Numerator = 0.0d0
        Denominator = 0.0d0
        Resistance = 0.0d0

        !'convenience variables
        ThisRadialCell  = ThisCell%PipeCellData%Soil(rCtr)
        InnerRadialCell = ThisCell%PipeCellData%Soil(rCtr - 1)
        OuterRadialCell = ThisCell%PipeCellData%Soil(rCtr + 1)

        !'any broadly defined variables
        Beta = ThisRadialCell%MyBase%Beta

        !'add effects from this cell history
        Numerator = Numerator + ThisRadialCell%MyBase%Temperature_PrevTimeStep
        Denominator = Denominator + 1

        !'add effects from outer cell
        Resistance = (LOG(OuterRadialCell%RadialCentroid / OuterRadialCell%InnerRadius) / &
                        (2 * Pi * Depth(ThisCell) * OuterRadialCell%MyBase%Properties%Conductivity)) &
                + (LOG(ThisRadialCell%OuterRadius/ThisRadialCell%RadialCentroid) / &
                        (2 * Pi * Depth(ThisCell) * ThisRadialCell%MyBase%Properties%Conductivity))
        Numerator = Numerator + (Beta / Resistance) * OuterRadialCell%MyBase%Temperature
        Denominator = Denominator + (Beta / Resistance)

        !'add effects from inner cell
        Resistance = (LOG(ThisRadialCell%RadialCentroid / ThisRadialCell%InnerRadius) / &
                        (2 * Pi * Depth(ThisCell) * ThisRadialCell%MyBase%Properties%Conductivity)) &
                + (LOG(InnerRadialCell%OuterRadius/InnerRadialCell%RadialCentroid) / &
                        (2 * Pi * Depth(ThisCell) * InnerRadialCell%MyBase%Properties%Conductivity))
        Numerator = Numerator + (Beta / Resistance) * InnerRadialCell%MyBase%Temperature
        Denominator = Denominator + (Beta / Resistance)

        !'calculate the new temperature
        ThisCell%PipeCellData%Soil(rCtr)%MyBase%Temperature = Numerator / Denominator

    END DO

END SUBROUTINE

SUBROUTINE SimulateInnerMostRadialSoilSlice(ThisCell)

    TYPE(CartesianCell), INTENT(IN OUT) :: ThisCell

    !'placeholder variables
    REAL(r64) :: Numerator
    REAL(r64) :: Denominator
    REAL(r64) :: Resistance
    REAL(r64) :: Beta

    TYPE(RadialCellInformation) :: ThisRadialCell
    TYPE(RadialCellInformation) :: InnerNeighborRadialCell
    TYPE(RadialCellInformation) :: OuterNeighborRadialCell

    Numerator = 0.0d0
    Denominator = 0.0d0

    !'convenience variables
    IF (Has%Insulation) THEN
        InnerNeighborRadialCell = ThisCell%PipeCellData%Insulation
    ELSE
        InnerNeighborRadialCell = ThisCell%PipeCellData%Pipe
    END IF
    ThisRadialCell = ThisCell%PipeCellData%Soil(0)
    OuterNeighborRadialCell = ThisCell%PipeCellData%Soil(1)

    !'any broadly defined variables
    Beta = ThisRadialCell%MyBase%Beta

    !'add effects from this cell history
    Numerator = Numerator + ThisRadialCell%MyBase%Temperature_PrevTimeStep
    Denominator = Denominator + 1

    !'add effects from outer radial cell
    Resistance = (LOG(OuterNeighborRadialCell%RadialCentroid / OuterNeighborRadialCell%InnerRadius) / &
                    (2 * PI * Depth(ThisCell) * OuterNeighborRadialCell%Mybase%Properties%Conductivity)) &
                +(LOG(ThisRadialCell%OuterRadius / ThisRadialCell%RadialCentroid) / &
                    (2 * PI * Depth(ThisCell) * ThisRadialCell%Mybase%Properties%Conductivity))
    Numerator = Numerator + (Beta / Resistance) * OuterNeighborRadialCell%MyBase%Temperature
    Denominator = Denominator + (Beta / Resistance)

    !'add effects from pipe cell
    Resistance = (LOG(ThisRadialCell%RadialCentroid / ThisRadialCell%InnerRadius) / &
                    (2 * PI * Depth(ThisCell) * ThisRadialCell%MyBase%Properties%Conductivity)) &
                + (LOG(InnerNeighborRadialCell%OuterRadius / InnerNeighborRadialCell%RadialCentroid) / &
                    (2 * PI * Depth(ThisCell) * InnerNeighborRadialCell%MyBase%Properties%Conductivity))
    Numerator = Numerator + (Beta / Resistance) * InnerNeighborRadialCell%MyBase%Temperature
    Denominator = Denominator + (Beta / Resistance)

    !'calculate the new temperature
    ThisCell%PipeCellData%Soil(0)%MyBase%Temperature = Numerator / Denominator
```

```
END SUBROUTINE

SUBROUTINE SimulateRadialInsulationCell(ThisCell)

    TYPE(CartesianCell), INTENT(IN OUT) :: ThisCell

    !'placeholder variables
    REAL(r64) :: Numerator
    REAL(r64) :: Denominator
    REAL(r64) :: Resistance
    REAL(r64) :: Beta

    TYPE(RadialCellInformation) :: PipeCell
    TYPE(RadialCellInformation) :: ThisInsulationCell
    TYPE(RadialCellInformation) :: NextInnerRadialCell

    Numerator = 0.0d0
    Denominator = 0.0d0

    !'convenience variables
    PipeCell = ThisCell%PipeCellData%Pipe
    ThisInsulationCell = ThisCell%PipeCellData%Insulation
    NextInnerRadialCell = ThisCell%PipeCellData%Soil(0)

    !'any broadly defined variables
    Beta = ThisInsulationCell%MyBase%Beta

    !'add effects from this cell history
    Numerator = Numerator + ThisInsulationCell%MyBase%Temperature_PrevTimeStep
    Denominator = Denominator + 1

    !'add effects from outer radial cell
    Resistance = (LOG(NextInnerRadialCell%RadialCentroid / NextInnerRadialCell%InnerRadius) / &
                    (2 * PI * Depth(ThisCell) * NextInnerRadialCell%MyBase%Properties%Conductivity)) &
                + (LOG(ThisInsulationCell%OuterRadius / ThisInsulationCell%RadialCentroid) / &
                    (2 * PI * Depth(ThisCell) * ThisInsulationCell%MyBase%Properties%Conductivity))
    Numerator = Numerator + (Beta / Resistance) * NextInnerRadialCell%MyBase%Temperature
    Denominator = Denominator + (Beta / Resistance)

    !'add effects from pipe cell
    Resistance = (LOG(ThisInsulationCell%RadialCentroid / ThisInsulationCell%InnerRadius) / &
                    (2 * PI * Depth(ThisCell) * ThisInsulationCell%MyBase%Properties%Conductivity)) &
                + (LOG(PipeCell%OuterRadius / PipeCell%RadialCentroid) / &
                    (2 * PI * Depth(ThisCell) * PipeCell%MyBase%Properties%Conductivity))
    Numerator = Numerator + (Beta / Resistance) * PipeCell%MyBase%Temperature
    Denominator = Denominator + (Beta / Resistance)

    !'calculate the new temperature
    ThisCell%PipeCellData%Insulation%MyBase%Temperature = Numerator / Denominator

END SUBROUTINE

SUBROUTINE SimulateRadialPipeCell(ThisCell, FlowRate, ConvectionCoefficient)

    TYPE(CartesianCell), INTENT(IN OUT) :: ThisCell
    REAL(r64), INTENT(IN) :: FlowRate
    REAL(r64), INTENT(IN) :: ConvectionCoefficient

    !'placeholder variables
    REAL(r64) :: Numerator
    REAL(r64) :: Denominator
    REAL(r64) :: Resistance
    REAL(r64) :: Beta
    REAL(r64) :: PipeConductionResistance
    REAL(r64) :: ConvectiveResistance

    TYPE(RadialCellInformation) :: ThisPipeCell
    TYPE(RadialCellInformation) :: OuterNeighborRadialCell
    TYPE(FluidCellInformation) :: FluidCell

    Numerator = 0.0d0
    Denominator = 0.0d0
    Resistance = 0.0d0

    !'convenience variables
    ThisPipeCell = ThisCell%PipeCellData%Pipe
    IF (Has%Insulation) THEN
        OuterNeighborRadialCell = ThisCell%PipeCellData%Insulation
    ELSE
        OuterNeighborRadialCell = ThisCell%PipeCellData%Soil(0)
    END IF
    FluidCell = ThisCell%PipeCellData%Fluid

    !'any broadly defined variables
    Beta = ThisPipeCell%MyBase%Beta

    !'add effects from this cell history
    Numerator = Numerator + ThisPipeCell%MyBase%Temperature_PrevTimeStep
    Denominator = Denominator + 1

    !'add effects from outer radial cell
```

```
        Resistance = (LOG(OuterNeighborRadialCell%RadialCentroid / OuterNeighborRadialCell%InnerRadius) / (2 * PI
              * Depth(ThisCell) * OuterNeighborRadialCell%MyBase%Properties%Conductivity)) &
           + (LOG(ThisPipeCell%OuterRadius / ThisPipeCell%RadialCentroid) / (2 * PI * Depth(ThisCell) *
                ThisPipeCell%MyBase%Properties%Conductivity))
        Numerator = Numerator + (Beta / Resistance) * OuterNeighborRadialCell%MyBase%Temperature
        Denominator = Denominator + (Beta / Resistance)

        !'add effects from water cell
        PipeConductionResistance = LOG(ThisPipeCell%RadialCentroid / ThisPipeCell%InnerRadius) / (2 * PI * Depth(
             ThisCell) * ThisPipeCell%MyBase%Properties%Conductivity)
        ConvectiveResistance = 1 / (ConvectionCoefficient * 2 * PI * ThisPipeCell%InnerRadius * Depth(ThisCell))
        Resistance = PipeConductionResistance + ConvectiveResistance
        Numerator = Numerator + (Beta / Resistance) * FluidCell%MyBase%Temperature
        Denominator = Denominator + (Beta / Resistance)

        !'calculate new temperature
        ThisCell%PipeCellData%Pipe%MyBase%Temperature = Numerator / Denominator

    END SUBROUTINE

    SUBROUTINE SimulateFluidCell(ThisCell, FlowRate, ConvectionCoefficient, EnteringFluidTemp)

        TYPE(CartesianCell), INTENT(IN OUT) :: ThisCell
        REAL(r64), INTENT(IN) :: FlowRate
        REAL(r64), INTENT(IN) :: ConvectionCoefficient
        REAL(r64), INTENT(IN) :: EnteringFluidTemp

        !'placeholder variables
        REAL(r64) :: Numerator
        REAL(r64) :: Denominator
        REAL(r64) :: TotalPipeResistance
        REAL(r64) :: PipeConductionResistance
        REAL(r64) :: ConvectiveResistance
        REAL(r64) :: UpstreamResistance
        REAL(r64) :: EnteringFluidConductance

        TYPE(FluidCellInformation) :: ThisFluidCell
        TYPE(RadialCellInformation) :: PipeCell

        Numerator = 0.0d0
        Denominator = 0.0d0

        !'convenience variables
        ThisFluidCell = ThisCell%PipeCellData%Fluid
        PipeCell = ThisCell%PipeCellData%Pipe

        !'add effects from this cell history
        Numerator = Numerator + ThisFluidCell%MyBase%Temperature_PrevTimeStep
        Denominator = Denominator + 1

        !'add effects from outer pipe cell
        PipeConductionResistance = LOG(PipeCell%RadialCentroid / PipeCell%InnerRadius) / &
                        (2 * PI * Depth(ThisCell) * PipeCell%MyBase%Properties%Conductivity)
        ConvectiveResistance = 1 / (ConvectionCoefficient * 2 * PI * PipeCell%InnerRadius * Depth(ThisCell))
        TotalPipeResistance = PipeConductionResistance + ConvectiveResistance
        Numerator = Numerator + (1 / TotalPipeResistance) * PipeCell%MyBase%Temperature
        Denominator = Denominator + (1 / TotalPipeResistance)

        !'add effects from upstream flow
        EnteringFluidConductance = 0.0d0
        IF (FlowRate > 0.0d0) THEN
            UpstreamResistance = 1 / (FlowRate * ThisFluidCell%Properties%MyBase%SpecificHeat)
            EnteringFluidConductance = ( (1/UpstreamResistance) - (0.5*TotalPipeResistance) )
            Numerator = Numerator + EnteringFluidConductance * EnteringFluidTemp
            Denominator = Denominator + EnteringFluidConductance
        END IF

        !'calculate new temperature
        ThisCell%PipeCellData%Fluid%MyBase%Temperature = Numerator / Denominator

    END SUBROUTINE

    SUBROUTINE DoOneTimeInitializations()

        INTEGER :: X, Y, Z, rCtr
        INTEGER :: NX, NY, NZ
        INTEGER :: SegCtr, SegIndex
        TYPE(PipeSegmentInfo) :: Segment
        INTEGER :: StartingZ
        INTEGER :: EndingZ
        INTEGER :: Increment
        INTEGER :: ZIndex
        INTEGER :: PipeX, PipeY
        INTEGER :: PrevUbound
        TYPE(Point3DInteger), ALLOCATABLE, DIMENSION(:) :: PrevEntries
        TYPE(CartesianCell) :: NeighborCell
        REAL(r64) :: NeighborTemp
        REAL(r64) :: Resistance
        REAL(r64) :: Beta
        TYPE(CartesianCell) :: ThisCell
        TYPE(RadialCellInformation) :: RadialCell
        INTEGER :: DirectionCtr
        INTEGER :: CurDirection
```

```
REAL(r64) :: Dummy = 0.0d0
INTEGER :: TotalSegments
INTEGER :: SegCtr2
REAL(r64) :: ThisCellTemp

!'initialize cell properties
DO Z = 0, UBOUND(Cells, 3)
    DO Y = 0, UBOUND(Cells, 2)
        DO X = 0, UBOUND(Cells, 1)

            SELECT CASE (Cells(X, Y, Z)%CellType)
            CASE (CellType_Pipe)
                Cells(X, Y, Z)%MyBase%Properties = GroundProperties
                DO rctr = 0, UBOUND(Cells(X, Y, Z)%PipeCellData%Soil, 1)
                    Cells(X, Y, Z)%PipeCellData%Soil(rctr)%MyBase%Properties = GroundProperties
                END DO
                Cells(X, Y, Z)%PipeCellData%Pipe%MyBase%Properties = PipeProperties
                IF (Has%Insulation) THEN
                    Cells(X, Y, Z)%PipeCellData%Insulation%MyBase%Properties = InsulationProperties
                END IF
            CASE (CellType_GeneralField, CellType_GroundSurface, CellType_AdiabaticWall,
                 CellType_FarfieldBoundary)
                Cells(X, Y, Z)%MyBase%Properties = GroundProperties
            CASE (CellType_BasementWall)
                Cells(X, Y, Z)%MyBase%Properties%Conductivity = BasementZone%BasementWall%MyBase%
                     Conductivity
                Cells(X, Y, Z)%MyBase%Properties%Density = BasementZone%BasementWall%MyBase%Density
                Cells(X, Y, Z)%MyBase%Properties%SpecificHeat = BasementZone%BasementWall%MyBase%
                     SpecificHeat
            CASE (CellType_BasementFloor, CellType_BasementCorner)
                Cells(X, Y, Z)%MyBase%Properties%Conductivity = BasementZone%BasementFloor%MyBase%
                     Conductivity
                Cells(X, Y, Z)%MyBase%Properties%Density = BasementZone%BasementFloor%MyBase%Density
                Cells(X, Y, Z)%MyBase%Properties%SpecificHeat = BasementZone%BasementFloor%MyBase%
                     SpecificHeat
            CASE (CellType_BasementCutaway)
                !shouldn't have to do anything...
            END SELECT

        END DO
    END DO
END DO

!'calculate one-time resistance terms for cartesian cells
DO Z = 0, UBOUND(Cells, 3)
    DO Y = 0, UBOUND(Cells, 2)
        DO X = 0, UBOUND(Cells, 1)
            CALL EvaluateCellNeighborDirections(Cells(X, Y, Z))
            DO DirectionCtr = 0, UBOUND(NeighborFieldCells,1)
                CurDirection = NeighborFieldCells(DirectionCtr)
                CALL EvaluateNeighborCharacteristics(Cells(X, Y, Z), CurDirection, NeighborTemp,
                     Resistance, NX, NY, NZ)
                CALL SetAdditionalNeighborData(X, Y, Z, CurDirection, Resistance, Cells(NX, NY, NZ))
            END DO
        END DO
    END DO
END DO

!'create circuit array for convenience
IF (Has%PipeCircuit) THEN

    SegCtr2 = -1

    TotalSegments = SIZE(Cells, 3) * SIZE(PipeCircuit%PipeSegments)
    ALLOCATE(PipeCircuit%ListOfCircuitPoints(0:TotalSegments-1))

    DO SegIndex = LBOUND(PipeCircuit%PipeSegments,1), UBOUND(PipeCircuit%PipeSegments,1)
        Segment = PipeCircuit%PipeSegments(SegIndex)

        !'set simulation flow direction
        SELECT CASE (Segment%FlowDirection)
        CASE (SegmentFlow_IncreasingZ)
            StartingZ = 0
            EndingZ = UBOUND(Cells,3)
            Increment = 1
        CASE (SegmentFlow_DecreasingZ)
            StartingZ = UBOUND(Cells,3)
            EndingZ = 0
            Increment = -1
        END SELECT

        PipeX = Segment%PipeCellCoordinates%X
        PipeY = Segment%PipeCellCoordinates%Y

        !'loop across all z-direction indeces
        DO Zindex = StartingZ, EndingZ, Increment
            SegCtr2 = SegCtr2 + 1
            PipeCircuit%ListOfCircuitPoints(SegCtr2) = Point3DInteger(PipeX, PipeY, Zindex)
        END DO

    END DO

END IF
```

```
    !'initialize freezing calculation variables
    CALL EvaluateSoilRhoCp(Dummy, Dummy, .TRUE.)

    !'we can also initialize the domain based on the farfield temperature here
    !likely not necessary to also init history terms here, but certainly cannot hurt!
    DO Z = 0, UBOUND(Cells, 3)
        DO Y = 0, UBOUND(Cells, 2)
            DO X = 0, UBOUND(Cells, 1)

                !On OneTimeInit, the cur sim time should be zero, so this will be OK
                ThisCellTemp = GetFarfieldTemp(Cells(X, Y, Z))
                Cells(X, Y, Z)%MyBase%Temperature = ThisCellTemp
                Cells(X, Y, Z)%MyBase%Temperature_PrevIteration = ThisCellTemp
                Cells(X, Y, Z)%MyBase%Temperature_PrevTimeStep = ThisCellTemp

                IF (Cells(X, Y, Z)%CellType == CellType_Pipe) THEN

                    DO rctr = 0, UBOUND(Cells(X, Y, Z)%PipeCellData%Soil, 1)
                        Cells(X, Y, Z)%PipeCellData%Soil(rctr)%MyBase%Temperature = ThisCellTemp
                        Cells(X, Y, Z)%PipeCellData%Soil(rctr)%MyBase%Temperature_PrevIteration =
                             ThisCellTemp
                        Cells(X, Y, Z)%PipeCellData%Soil(rctr)%MyBase%Temperature_PrevTimeStep = ThisCellTemp
                    END DO
                    Cells(X, Y, Z)%PipeCellData%Pipe%MyBase%Temperature = ThisCellTemp
                    Cells(X, Y, Z)%PipeCellData%Pipe%MyBase%Temperature_PrevIteration = ThisCellTemp
                    Cells(X, Y, Z)%PipeCellData%Pipe%MyBase%Temperature_PrevTimeStep = ThisCellTemp
                    IF (Has%Insulation) THEN
                        Cells(X, Y, Z)%PipeCellData%Insulation%MyBase%Temperature = ThisCellTemp
                        Cells(X, Y, Z)%PipeCellData%Insulation%MyBase%Temperature_PrevIteration =
                             ThisCellTemp
                        Cells(X, Y, Z)%PipeCellData%Insulation%MyBase%Temperature_PrevTimeStep = ThisCellTemp
                    END IF
                    Cells(X, Y, Z)%PipeCellData%Fluid%MyBase%Temperature = ThisCellTemp
                    Cells(X, Y, Z)%PipeCellData%Fluid%MyBase%Temperature_PrevIteration = ThisCellTemp
                    Cells(X, Y, Z)%PipeCellData%Fluid%MyBase%Temperature_PrevTimeStep = ThisCellTemp

                END IF

            END DO
        END DO
    END DO

END SUBROUTINE

SUBROUTINE DoStartOfTimeStepInitializations()

    !TYPE(CartesianCell), POINTER :: CellToCheck
    !TYPE(RadialCellInformation), POINTER :: PipeCell
    TYPE(RadialCellInformation) :: RadialCell
    !TYPE(FluidCellInformation), POINTER :: WaterCell
    REAL(r64) :: EnteringTemp
    REAL(r64) :: ExitingTemp
    REAL(r64) :: AverageTemp
    REAL(r64) :: Density
    REAL(r64) :: Viscosity
    REAL(r64) :: Conductivity
    REAL(r64) :: SpecificHeat
    REAL(r64) :: Prandtl
    REAL(r64) :: Area_c
    REAL(r64) :: Velocity
    REAL(r64) :: Reynolds
    REAL(r64) :: Exponent
    REAL(r64) :: Nusselt
    REAL(r64) :: ConvCoefficient
    INTEGER :: X, Y, Z
    REAL(r64) :: Temperature
    REAL(r64) :: Beta
    REAL(r64) :: CellTemp
    REAL(r64) :: CellRhoCp
    INTEGER :: radialctr
    INTEGER :: rCtr

    !Call this to update current sim time, time step size, etc.
    CALL UpdateTransientConditions()

    !'pipe circuit conditions
    IF (Has%PipeCircuit) THEN
        PipeCircuit%CurFluidPropertySet = ExtendedFluidProperties( BaseThermalPropertySet(
             CurFluidConductivity, CurFluidDensity, CurFluidSpecificHeat), CurFluidViscosity,
             CurFluidViscosity)
    END IF

    !'now update cell properties
    !IF (Has%Moisture) THEN
    IF (DoingFreezing) THEN
        DO Z = LBOUND(Cells,3), UBOUND(Cells,3)
            DO Y = LBOUND(Cells,2), UBOUND(Cells,2)
                DO X = LBOUND(Cells,1), UBOUND(Cells,1)
                    !'since the cell properties are instantiated separately, we can now just set the Cp value
                          easily here without all the reallocation hackyness
                    SELECT CASE(Cells(X, Y, Z)%CellType)
                    CASE(CellType_GeneralField, CellType_AdiabaticWall, CellType_FarfieldBoundary,
                         CellType_GroundSurface)
                        !'main ground cells, update with soil properties
                        CellTemp = Cells(X, Y, Z)%MyBase%Temperature
                        CALL EvaluateSoilRhoCp(CellTemp, CellRhoCp)
                        Cells(X, Y, Z)%MyBase%Properties%SpecificHeat = CellRhoCp / Cells(X, Y, Z)%MyBase%
                             Properties%Density
                    CASE(CellType_BasementCorner, CellType_BasementCutAway, CellType_BasementFloor,
                         CellType_BasementWall)
                        !'basement cells, for now they have constant properties
                    CASE(CellType_Pipe)
                        !'first update the outer cell itself
                        CellTemp = Cells(X, Y, Z)%MyBase%Temperature
                        CALL EvaluateSoilRhoCp(CellTemp, CellRhoCp)
                        Cells(X, Y, Z)%MyBase%Properties%SpecificHeat = CellRhoCp / Cells(X, Y, Z)%MyBase%
                             Properties%Density
                        !'then update all the soil radial cells
                        DO radialctr = LBOUND(Cells(X,Y,Z)%PipeCellData%Soil,1), UBOUND(Cells(X,Y,Z)%
                             PipeCellData%Soil,1)
                            CellTemp = Cells(X, Y, Z)%PipeCellData%Soil(radialctr)%MyBase%Temperature
                            CALL EvaluateSoilRhoCp(CellTemp, CellRhoCp)
                            Cells(X, Y, Z)%PipeCellData%Soil(radialctr)%MyBase%Properties%SpecificHeat =
                                 CellRhoCp / Cells(X, Y, Z)%PipeCellData%Soil(radialctr)%MyBase%Properties%
                                 Density
                        END DO
                    END SELECT
                END DO
            END DO
        END DO
    END IF

    !'calculate the beta values for all cells
    DO Z = 0, UBOUND(Cells, 3)
        DO Y = 0, UBOUND(Cells, 2)
            DO X = 0, UBOUND(Cells, 1)

                SELECT CASE (Cells(X, Y, Z)%CellType)
                CASE (CellType_Pipe)

                    !'set the interface cell
                    Beta = CurSimTimeStepSize / (Cells(X, Y, Z)%MyBase%Properties%Density * Cells(X, Y, Z)%
                         PipeCellData%InterfaceVolume * Cells(X, Y, Z)%MyBase%Properties%SpecificHeat)
                    Cells(X, Y, Z)%MyBase%Beta = Beta

                    !'set the radial soil cells
                    DO rctr = 0, UBOUND(Cells(X, Y, Z)%PipeCellData%Soil,1)
                        RadialCell = Cells(X, Y, Z)%PipeCellData%Soil(rctr)
                        Beta = CurSimTimeStepSize / (RadialCell%MyBase%Properties%Density *
                             RadialCellInfo_XY_CrossSectArea(RadialCell) * Depth(Cells(X, Y, Z)) * RadialCell
                             %MyBase%Properties%SpecificHeat)
                        Cells(X, Y, Z)%PipeCellData%Soil(rctr)%MyBase%Beta = Beta
                    END DO

                    !'then insulation if it exists
                    IF (Has%Insulation) THEN
                        RadialCell = Cells(X, Y, Z)%PipeCellData%Insulation
                        Beta = CurSimTimeStepSize / (RadialCell%MyBase%Properties%Density *
                             RadialCellInfo_XY_CrossSectArea(RadialCell) * Depth(Cells(X, Y, Z)) * RadialCell
                             %MyBase%Properties%SpecificHeat)
                        Cells(X, Y, Z)%PipeCellData%Insulation%MyBase%Beta = Beta
                    END IF

                    !'set the pipe cell
                    RadialCell = Cells(X, Y, Z)%PipeCellData%Pipe
                    Beta = CurSimTimeStepSize / (RadialCell%MyBase%Properties%Density *
                         RadialCellInfo_XY_CrossSectArea(RadialCell) * Depth(Cells(X, Y, Z)) * RadialCell%
                         MyBase%Properties%SpecificHeat)
                    Cells(X, Y, Z)%PipeCellData%Pipe%MyBase%Beta = Beta

                    !'since water cells now have variable properties, we need to init their values during
                         each time step, not "one-time" here

                    IF (Has%PipeCircuit) THEN
                        IF (Cells(X, Y, Z)%CellType == CellType_Pipe) THEN
                            Temperature = Cells(X, Y, Z)%PipeCellData%Fluid%MyBase%Temperature
                            Cells(X, Y, Z)%PipeCellData%Fluid%Properties = PipeCircuit%CurFluidPropertySet
                            Cells(X, Y, Z)%PipeCellData%Fluid%MyBase%Beta = &
                                CurSimTimeStepSize / (Cells(X, Y, Z)%PipeCellData%Fluid%Properties%MyBase%
                                     Density * Cells(X, Y, Z)%PipeCellData%Fluid%Volume * Cells(X, Y, Z)%
                                     PipeCellData%Fluid%Properties%MyBase%SpecificHeat)
                        END IF
                    END IF

                CASE (CellType_BasementCutAway)
                    !'this area is not calculated either, so no properties

                CASE DEFAULT
                    !'these are basic cartesian calculation cells
                    Beta = CurSimTimeStepSize / (Cells(X, Y, Z)%MyBase%Properties%Density * Volume(Cells(X, Y
                         , Z)) * Cells(X, Y, Z)%MyBase%Properties%SpecificHeat)
                    Cells(X, Y, Z)%MyBase%Beta = Beta

                END SELECT
            END DO
        END DO
    END DO

END SUBROUTINE

SUBROUTINE DoEndOfIterationOperations(IterationIndex, Finished, ErrorsFound)

    INTEGER, INTENT(IN) :: IterationIndex
    LOGICAL, INTENT(IN OUT) :: Finished
    LOGICAL, INTENT(IN OUT) :: ErrorsFound

    REAL(r64) :: MaxDivergence_FromLastIteration
    TYPE(CartesianCell) :: MaxUnconvergedCell
    LOGICAL :: Converged_FromLastIteration
    LOGICAL :: OutOfRange

    !'check if we have converged for this iteration if we are doing implicit transient
    Converged_FromLastIteration = IsConverged_CurrentToPrevIteration(MaxDivergence_FromLastIteration)

    !'check for out of range temperatures here so they aren't plotted
    !'this routine should be *much* more restrictive than the exceptions, so we should be safe with this
         location
    OutOfRange = CheckForOutOfRangeTemps()
    IF (OutOfRange) THEN
        ErrorsFound = .TRUE.
    END IF

    !'if we are doing implicit transient and we are converged in the iteration loop then we can leave the
         iteration loop
    IF (Converged_FromLastIteration) THEN
        Finished = .TRUE.
    END IF

END SUBROUTINE

SUBROUTINE EvaluateSoilRhoCp(CellTemp, rhoCp, InitOnly)

    REAL(r64), INTENT(IN) :: CellTemp
    REAL(r64), INTENT(OUT) :: rhoCp
    LOGICAL, INTENT(IN), OPTIONAL :: InitOnly

    !'static variables only calculated once per simulation run
    REAL(r64), SAVE ::  Theta_ice
    REAL(r64), SAVE ::  Theta_liq
    REAL(r64), SAVE ::  Theta_sat
    REAL(r64), SAVE ::  rho_ice
    REAL(r64), SAVE ::  rho_liq
    REAL(r64), SAVE ::  rhoCp_soil_liq_1
    REAL(r64), SAVE ::  CP_liq
    REAL(r64), SAVE ::  CP_ice
    REAL(r64), SAVE ::  Lat_fus
    REAL(r64), SAVE ::  Cp_transient
    REAL(r64), SAVE ::  rhoCP_soil_liq
    REAL(r64), SAVE ::  rhoCP_soil_transient
    REAL(r64), SAVE ::  rhoCP_soil_ice

    REAL(r64) :: frzAllIce
    REAL(r64) :: frzIceTrans
    REAL(r64) :: frzLiqTrans
    REAL(r64) :: frzAllLiq
    REAL(r64) :: rhoCP_soil

    IF (PRESENT(InitOnly)) THEN
        !'Cp (freezing) calculations
        Theta_ice = 0.3
        Theta_liq = 0.3 !'moisture content of the soil
        Theta_sat = 0.5
        rho_ice = 917 !'Kg / m3
        rho_liq = 1000 !'kg / m3
        rhoCp_soil_liq_1 = 1225000.0 / (1 - Theta_sat) !'J/m3K
        !'from(" An improved model for predicting soil thermal conductivity from water content at room
             temperature, Fig 4")
        CP_liq = 4180.0 !'J / KgK
        CP_ice = 2066.0 !'J / KgK
        Lat_fus = 334000 !'J / Kg
        Cp_transient = Lat_fus / 0.4 + (0.5 * CP_ice - (CP_liq + CP_ice) / 2 * 0.1) / 0.4
        !'from(" Numerical and experimental investigation of melting and freezing processes in phase change
             material storage")
        rhoCP_soil_liq = rhoCp_soil_liq_1 * (1 - Theta_sat) + rho_liq * CP_liq * Theta_liq
        rhoCP_soil_transient = rhoCp_soil_liq_1 * (1 - Theta_sat) + ((rho_liq + rho_ice)/2.0d0) *
             Cp_transient * Theta_ice
        rhoCP_soil_ice = rhoCp_soil_liq_1 * (1 - Theta_sat) + rho_ice * CP_ice * Theta_ice  !'!J / m3K
        RETURN
    END IF

    !'set some temperatures here for generalization -- these will be set in the input file
    frzAllIce = -0.5
    frzIceTrans = -0.4
    frzLiqTrans = -0.1
    frzAllLiq = 0.0
    !'calculate this cell's new Cp value based on the cell temperature
    IF (CellTemp >= frzAllLiq) THEN
        rhoCP_soil = rhoCp_soil_liq_1
    ELSE IF (CellTemp <= frzAllIce) THEN
        rhoCP_soil = rhoCP_soil_ice
    ELSE IF ((CellTemp < frzAllLiq) .AND. (CellTemp > frzLiqTrans)) THEN
        rhoCP_soil = rhoCp_soil_liq_1 + (rhoCP_soil_transient - rhoCP_soil_liq) / (frzAllLiq - frzLiqTrans) *
              (frzAllLiq - CellTemp)
    ELSE IF ((CellTemp <= frzLiqTrans) .AND. (CellTemp >= frzIceTrans)) THEN
        rhoCP_soil = rhoCP_soil_transient
    ELSE IF ((CellTemp < frzIceTrans) .AND. (CellTemp > frzAllIce)) THEN
        rhoCP_soil = rhoCP_soil_transient + (rhoCP_soil_transient - rhoCP_soil_ice) / (frzIceTrans -
             frzAllIce) * (CellTemp - frzAllIce)
    End If
    rhoCp = rhoCP_soil

END SUBROUTINE

SUBROUTINE SetAdditionalNeighborData(X, Y, Z, Direction, Resistance, NeighborCell)

    INTEGER, INTENT(IN) :: X, Y, Z
    INTEGER, INTENT(IN) :: Direction
    REAL(r64), INTENT(IN) :: Resistance
    TYPE(CartesianCell), INTENT(IN) :: NeighborCell

    INTEGER :: NeighborIndex

    DO NeighborIndex = 0, UBOUND(Cells(X, Y, Z)%NeighborInformation,1)
        IF (Cells(X, Y, Z)%NeighborInformation(NeighborIndex)%Direction == Direction) THEN
            Cells(X, Y, Z)%NeighborInformation(NeighborIndex)%Value%ConductionResistance = Resistance
            Cells(X, Y, Z)%NeighborInformation(NeighborIndex)%Value%NeighborCellIndeces = Point3DInteger(
                 NeighborCell%X_index, NeighborCell%Y_index, NeighborCell%Z_index)
        END IF
    END DO

END SUBROUTINE

SUBROUTINE EvaluateNeighborCharacteristics(ThisCell, CurDirection, NeighborTemp, Resistance, NeighborX,
     NeighborY, NeighborZ)

    TYPE(CartesianCell), INTENT(IN) :: ThisCell
    INTEGER, INTENT(IN) :: CurDirection
    REAL(r64), INTENT(OUT) :: NeighborTemp
    REAL(r64), INTENT(OUT) :: Resistance
    INTEGER, INTENT(OUT), OPTIONAL :: NeighborX
    INTEGER, INTENT(OUT), OPTIONAL :: NeighborY
    INTEGER, INTENT(OUT), OPTIONAL :: NeighborZ

    REAL(r64) :: ThisCellLength
    REAL(r64) :: NeighborCellLength
    REAL(r64) :: ThisCellConductivity
    REAL(r64) :: NeighborConductivity
    REAL(r64) :: ThisNormalArea
    REAL(r64) :: ConvectiveResistance
    TYPE(NeighborInformation) :: TempNeighborInfo
    INTEGER :: CellWidthsUbound

    INTEGER :: NX, NY, NZ
    INTEGER :: X, Y, Z

    X = ThisCell%X_index
    Y = ThisCell%Y_index
    Z = ThisCell%Z_index

    !'get neighbor data
    SELECT CASE (CurDirection)
    CASE (Direction_PositiveY)
        NX = X
        NY = Y + 1
        NZ = Z
    CASE (Direction_NegativeY)
        NX = X
        NY = Y - 1
        NZ = Z
    CASE (Direction_PositiveX)
        NX = X + 1
        NY = Y
        NZ = Z
    CASE (Direction_NegativeX)
        NX = X - 1
        NY = Y
        NZ = Z
    CASE (Direction_PositiveZ)
        NX = X
        NY = Y
        NZ = Z + 1
    CASE (Direction_NegativeZ)
        NX = X
        NY = Y
        NZ = Z - 1
    END SELECT

    !'split effects between the two cells so we can carefully calculate resistance values
    ThisCellLength = 0.0d0
    NeighborCellLength = 0.0d0
    ThisCellConductivity = HUGE(1.0d0)
    IF (ThisCell%MyBase%Properties%Conductivity > 0.0d0) ThisCellConductivity = ThisCell%MyBase%Properties%
         Conductivity
    NeighborConductivity = HUGE(1.0d0)
    IF (Cells(NX, NY, NZ)%MyBase%Properties%Conductivity > 0.0d0) NeighborConductivity = Cells(NX, NY, NZ)%
         MyBase%Properties%Conductivity

    !'calculate normal surface area
    ThisNormalArea = NormalArea(ThisCell, CurDirection)

    !'set distance based on cell types
    TempNeighborInfo = NeighborInformationArray_Value(ThisCell%NeighborInformation, CurDirection)
    IF (ThisCell%CellType == CellType_Pipe) THEN
        !'we need to be a bit careful with pipes, as they are full centroid to centroid in the z direction,
        !' but only centroid to wall in the x and y directions
        IF (CurDirection == Direction_NegativeZ .OR. CurDirection == Direction_PositiveZ) THEN
            ThisCellLength = TempNeighborInfo%ThisCentroidToNeighborWall
            NeighborCellLength = TempNeighborInfo%ThisWallToNeighborCentroid
        ELSE
            ThisCellLength = 0
            NeighborCellLength = TempNeighborInfo%ThisWallToNeighborCentroid
        END IF
    ELSE IF (Cells(NX, NY, NZ)%CellType == CellType_Pipe) THEN
        ThisCellLength = TempNeighborInfo%ThisCentroidToNeighborWall
        NeighborCellLength = 0
    ELSE IF (Cells(NX, NY, NZ)%CellType == CellType_BasementCutAway) THEN
        ThisCellLength = TempNeighborInfo%ThisCentroidToNeighborWall
        NeighborCellLength = 0
    ELSE
        ThisCellLength = TempNeighborInfo%ThisCentroidToNeighborWall
        NeighborCellLength = TempNeighborInfo%ThisWallToNeighborCentroid
    END IF

    !'also set any convective resistance
    ConvectiveResistance = 0.0d0
    IF (Cells(NX, NY, NZ)%CellType == CellType_BasementCutAway) THEN
        IF (ThisCell%CellType == CellType_BasementWall) THEN
            ConvectiveResistance = 1 / (BasementWallConvCoeff * ThisNormalArea)
        ELSE IF (ThisCell%CellType == CellType_BasementFloor) THEN
            ConvectiveResistance = 1 / (BasementFloorConvCoeff * ThisNormalArea)
        END IF
    END IF

    !'calculate resistance based on different conductivities between the two cells
    Resistance = (ThisCellLength / (ThisNormalArea * ThisCellConductivity)) + &
     (NeighborCellLength / (ThisNormalArea * NeighborConductivity)) + &
     ConvectiveResistance

    !'return proper temperature for the given simulation type
    IF (Cells(NX, NY, NZ)%CellType == CellType_BasementCutAway) THEN
        NeighborTemp = BasementTemp
    ELSE
        NeighborTemp = Cells(NX, NY, NZ)%MyBase%Temperature
    END IF

    IF (PRESENT(NeighborX)) THEN
        NeighborX = NX
        NeighborY = NY
        NeighborZ = NZ
    END IF

END SUBROUTINE

SUBROUTINE EvaluateCellNeighborDirections(cell)

    TYPE(CartesianCell), INTENT(IN) :: cell
    INTEGER :: Xmax, Ymax, Zmax
    INTEGER :: Xindex, Yindex, Zindex
    INTEGER :: NumFieldCells, NumBoundaryCells
    INTEGER :: FieldCellCtr, BoundaryCellCtr
    INTEGER, PARAMETER :: TotalNumDimensions = 6

    Xmax = UBOUND(Cells,1)
    Ymax = UBOUND(Cells,2)
    Zmax = UBOUND(Cells,3)
    Xindex = cell%X_index
    Yindex = cell%Y_index
    Zindex = cell%Z_index
    !Initialize the counters

    NumFieldCells = 0
    NumBoundaryCells = 0

    !First get the count for each array
    IF(Xindex < Xmax)  NumFieldCells = NumFieldCells + 1
    IF(Xindex > 0)     NumFieldCells = NumFieldCells + 1
    IF(Yindex < Ymax)  NumFieldCells = NumFieldCells + 1
    IF(Yindex > 0)     NumFieldCells = NumFieldCells + 1
    IF(Zindex < Zmax)  NumFieldCells = NumFieldCells + 1
    IF(Zindex > 0)     NumFieldCells = NumFieldCells + 1
    NumBoundaryCells = TotalNumDimensions - NumFieldCells
```

```
        !Allocate the arrays
        IF (ALLOCATED(NeighborFieldCells)) DEALLOCATE(NeighborFieldCells)
        ALLOCATE(NeighborFieldCells(0:NumFieldCells-1))
        IF (ALLOCATED(NeighborBoundaryCells)) DEALLOCATE(NeighborBoundaryCells)
        ALLOCATE(NeighborBoundaryCells(0:NumBoundaryCells-1))

        !Then add to each array appropriately
        FieldCellCtr = -1
        BoundaryCellCtr = -1
        IF(Xindex < Xmax) THEN
            FieldCellCtr = FieldCellCtr + 1
            NeighborFieldCells(FieldCellCtr) = Direction_PositiveX
        ELSE
            BoundaryCellCtr = BoundaryCellCtr + 1
            NeighborBoundaryCells(BoundaryCellCtr) = Direction_PositiveX
        END IF

        IF(Xindex > 0)    THEN
            FieldCellCtr = FieldCellCtr + 1
            NeighborFieldCells(FieldCellCtr) = Direction_NegativeX
        ELSE
            BoundaryCellCtr = BoundaryCellCtr + 1
            NeighborBoundaryCells(BoundaryCellCtr) = Direction_NegativeX
        END IF

        IF(Yindex < Ymax) THEN
            FieldCellCtr = FieldCellCtr + 1
            NeighborFieldCells(FieldCellCtr) = Direction_PositiveY
        ELSE
            BoundaryCellCtr = BoundaryCellCtr + 1
            NeighborBoundaryCells(BoundaryCellCtr) = Direction_PositiveY
        END IF

        IF(Yindex > 0)    THEN
            FieldCellCtr = FieldCellCtr + 1
            NeighborFieldCells(FieldCellCtr) = Direction_NegativeY
        ELSE
            BoundaryCellCtr = BoundaryCellCtr + 1
            NeighborBoundaryCells(BoundaryCellCtr) = Direction_NegativeY
        END IF

        IF(Zindex < Zmax) THEN
            FieldCellCtr = FieldCellCtr + 1
            NeighborFieldCells(FieldCellCtr) = Direction_PositiveZ
        ELSE
            BoundaryCellCtr = BoundaryCellCtr + 1
            NeighborBoundaryCells(BoundaryCellCtr) = Direction_PositiveZ
        END IF

        IF(Zindex > 0)    THEN
            FieldCellCtr = FieldCellCtr + 1
            NeighborFieldCells(FieldCellCtr) = Direction_NegativeZ
        ELSE
            BoundaryCellCtr = BoundaryCellCtr + 1
            NeighborBoundaryCells(BoundaryCellCtr) = Direction_NegativeZ
        END IF

    END SUBROUTINE

    SUBROUTINE UpdateTransientConditions()

!Timestep   Dry Bulb    RH  Wind Speed  Solar Rad

        TYPE TransientDataPoint
            REAL(r64) :: TimeStamp !Seconds
            !REAL(r64) :: CircuitFlowRate !not used in UGT validation
            REAL(r64) :: DryBulb
            !REAL(r64) :: CircuitEFT !not used in UGT validation
            !REAL(r64) :: BasementAirTemp !not used in UGT validation
            REAL(r64) :: RelativeHumidity
            REAL(r64) :: WindSpeed
            REAL(r64) :: IncidentSolar
            !REAL(r64) :: CircuitHPHeatAddedToFluid !not used in validation
        END TYPE

        LOGICAL :: EndOfFile
        INTEGER :: Pos
        INTEGER :: LineLength
        LOGICAL, SAVE :: OneTimeInit = .TRUE.
        CHARACTER(LEN=200) ReadLine
        CHARACTER(LEN=50), DIMENSION(5) :: Tokens
        INTEGER :: PrevUbound
        INTEGER :: IOStatus
        INTEGER :: DataPointCtr
        INTEGER :: PreviousTimeStamp
        REAL(r64) :: CurTimeStamp

        TYPE(TransientDataPoint), DIMENSION(0:40000) :: TempTransientData
        TYPE(TransientDataPoint), ALLOCATABLE, DIMENSION(:), SAVE :: TransientData
        TYPE(TransientDataPoint) :: TempDataPoint

        !The first time through we need to get the transient data from the file
        IF (OneTimeInit) THEN

            WRITE(*,*) 'Reading␣TransientData_UGT.csv␣data␣file...'
            OPEN(81, FILE='TransientData_UGT.csv', ERR=2980)

            EndOfFile = .FALSE.
            DataPointCtr = -1

            DO WHILE (.NOT. EndOfFile)

                READ(81, '(A)', IOSTAT=IOStatus) ReadLine

                IF (IOStatus .NE. 0) EXIT

                IF (LEN_TRIM(ADJUSTL(ReadLine)) > 1) THEN
                    IF (ReadLine(1:1)=="!" .OR. ReadLine(2:2)=="!") CYCLE
                END IF

                DataPointCtr = DataPointCtr + 1

                !Read TimeStamp
                ReadLine = ADJUSTL(ReadLine)
                Pos = SCAN(ReadLine, ',')
                READ(ReadLine(1:Pos-1), '(A)') Tokens(1)
                LineLength = LEN(ReadLine)
                ReadLine = ReadLine(Pos+1:LineLength)

                !Read Dry Bulb
                ReadLine = ADJUSTL(ReadLine)
                Pos = SCAN(ReadLine, ',')
                READ(ReadLine(1:Pos-1), '(A)') Tokens(2)
                LineLength = LEN(ReadLine)
                ReadLine = ReadLine(Pos+1:LineLength)

                !Read Relative Humidity
                ReadLine = ADJUSTL(ReadLine)
                Pos = SCAN(ReadLine, ',')
                READ(ReadLine(1:Pos-1), '(A)') Tokens(3)
                LineLength = LEN(ReadLine)
                ReadLine = ReadLine(Pos+1:LineLength)

                !Read Wind Speed
                ReadLine = ADJUSTL(ReadLine)
                Pos = SCAN(ReadLine, ',')
                READ(ReadLine(1:Pos-1), '(A)') Tokens(4)
                LineLength = LEN(ReadLine)
                ReadLine = ReadLine(Pos+1:LineLength)

!                !Read Circuit Flow Rate
!                ReadLine = ADJUSTL(ReadLine)
!                Pos = SCAN(ReadLine, ',')
!                READ(ReadLine(1:Pos-1), '(A)') Tokens(5)
!                LineLength = LEN(ReadLine)
!                ReadLine = ReadLine(Pos+1:LineLength)
!
!                !Read Circuit HP Q (Heat added to FHX fluid)
!                ReadLine = ADJUSTL(ReadLine)
!                Pos = SCAN(ReadLine, ',')
!                READ(ReadLine(1:Pos-1), '(A)') Tokens(6)
!                LineLength = LEN(ReadLine)
!                ReadLine = ReadLine(Pos+1:LineLength)

                !Read Solar Radiation
                ReadLine = ADJUSTL(ReadLine)
                READ(ReadLine, '(A)') Tokens(5)

                !Now process all the tokens into numeric data
                READ(Tokens(1), *) TempDataPoint%TimeStamp
                READ(Tokens(2), *) TempDataPoint%DryBulb
                READ(Tokens(3), *) TempDataPoint%RelativeHumidity
                READ(Tokens(4), *) TempDataPoint%WindSpeed
                READ(Tokens(5), *) TempDataPoint%IncidentSolar
                !READ(Tokens(6), *) TempDataPoint%CircuitHPHeatAddedToFluid
                !READ(Tokens(7), *) TempDataPoint%BasementAirTemp

                TempTransientData(DataPointCtr) = TempDataPoint
!                IF (.NOT. ALLOCATED(TransientData)) THEN
!                    ALLOCATE(TransientData(0:0))
!                    TransientData(0) = TempDataPoint
!                ELSE
!                    PrevUbound = UBOUND(TransientData, 1)
!                    IF (ALLOCATED(TempTransientData)) DEALLOCATE(TempTransientData)
!                    ALLOCATE(TempTransientData(0:PrevUbound))
!                    TempTransientData(0:PrevUbound) = TransientData
!                    DEALLOCATE(TransientData)
!                    ALLOCATE(TransientData(0:PrevUbound+1))
!                    TransientData(0:PrevUbound) = TempTransientData
!                    TransientData(PrevUbound+1) = TempDataPoint
!                    DEALLOCATE(TempTransientData)
!                END IF

                !IF (REAL(SIZE(TransientData))/250.0d0 == INT(REAL(SIZE(TransientData))/250.0d0)) THEN
                !    WRITE(*, *) "Found ", DataPointCtr, " data points ... so far ..."
                !END IF

            END DO

            IF (ALLOCATED(TransientData)) DEALLOCATE(TransientData) !Should NOT be necessary, but safe
            ALLOCATE(TransientData(0:DataPointCtr))
            TransientData = TempTransientData(0:DataPointCtr)

            !we are done reading, make sure we close the file if it is open
2950        CLOSE(81)
            WRITE(*, *) "Interpreted␣", SIZE(TransientData), "␣data␣points"

            OneTimeInit = .FALSE.
        END IF

        !Interpolate and set weather conditions
        PreviousTimeStamp = 0.0d0
        DO DataPointCtr = LBOUND(TransientData,1), UBOUND(TransientData,1)
            TempDataPoint = TransientData(DataPointCtr)
            CurTimeStamp = TempDataPoint%TimeStamp
            IF ((CurSimTimeSeconds > PreviousTimeStamp .AND. CurSimTimeSeconds <= CurTimeStamp) &
                  .OR. (DataPointCtr == UBOUND(TransientData,1))) THEN
                CurAirTemp = TempDataPoint%DryBulb
                CurWindSpeed = TempDataPoint%WindSpeed
                CurRelativeHumidity = TempDataPoint%RelativeHumidity
                CurIncidentSolar = TempDataPoint%IncidentSolar
                !CurCircuitFlowRate = TempDataPoint%CircuitFlowRate
                !BasementTemp = TempDataPoint%BasementAirTemp
                !CurCircuitHeatPumpQ = TempDataPoint%CircuitHPHeatAddedToFluid
                EXIT
            END IF
            PreviousTimeStamp = CurTimeStamp
        END DO

        RETURN

!ErrHandler
2980 CONTINUE
        WRITE(*, '(A100)') 'Could␣not␣find␣TransientData.csv...aborting...press␣ENTER␣to␣exit...'
        READ(*, *)
        STOP

    END SUBROUTINE

END MODULE
```

APPENDIX C

Transport Delay Testbed Source

Listing C.1: Transport Delay Testbed Source: Manager

```
!!This program will perform varying levels of transport delay calculations
! This work authored by Edwin Lee, at Oklahoma State University
!  ___  _     ____  _       _
! / _ \| | __/ ___|| |_ __ _| |_ ___
!| | | | |/ /\___ \| __/ _` | __/ _ \
!| |_| |   <  ___) | || (_| | ||  __/
! \___/|_|\_\|____/ \__\__,_|\__\___|

!ToDo:  improved boundary condition spec, not just pipe outer temperature...or maybe this is OK
!       laminar & heating/cooling nusselt correlations

!Info:  implementing time variant flow would require variable segment length, so this is not done
!       well-mixed adiabatic demonstrates the inherent steady state nature of the well-mixed model

! There are basically three features to transport delay:
!  - mixing within a segment
!  - heat transfer from the segment to the boundary condition
!  - and transient storage within the segment

! begin program!
program transportdelay

! use the enumeration definitions
use enumerations

! access any utilities
use transportdelay_utilities

! access data structure
use structures

! enforce explicit declarations
implicit none

! ================================================== |
! = = = VARIABLES DERIVED DIRECTLY FROM INPUTS = = = |
! -------------------------------------------------- |
!~TYPE~~||~~~~~~~~~VAR NAME~~~~~~~~~~|~~~~UNITS~~~~~|
! -------------------------------------------------- |
real    :: segment_length             ! [m] --------- |
real    :: pipe_cross_sectional_area  ! [m2] -------- |
real    :: fluid_volume_flow_rate     ! [m3/s] ------ |
real    :: fluid_velocity             ! [m/s] ------- |
real    :: segment_residence_time     ! [s] --------- |
real    :: pipe_residence_time        ! [s] --------- |
real    :: time_step                  ! [s] --------- |
integer :: num_time_steps             ! [-] --------- |
real    :: segment_surface_area       ! [m2] -------- |
real    :: segment_out_surface_area   ! [m2] -------- |
real    :: fluid_mass_in_a_segment    ! [kg] -------- |
real    :: time_heatpump_cutoff       ! [s] --------- |
! ================================================== |

! = = = = = = = = = = = = = = GENERAL VARIABLES = = = = = = = = = = = = = = = = = = |
!~~~~~~~~VAR TYPE~~~~~~~~~||~~~~~~~~VAR NAME~~~~~~~~~~~~~|~~INITVAL~~|~~~UNITS~~~~~|
type(fluid_segment), & ! ---------------------------------------------------------- |
 dimension(:), allocatable :: fluid_segments                       ! ----------- | (main variable: array of
      fluid segments)
real, & ! --------------------------------------------------------- |
 dimension(:), allocatable :: segmentTemps_prevTime              ! ----------- | (main variable: array of
      fluid segment temps at previous time step)
integer                    :: fluid_segment_index                ! [-] ------- | (counter for segment loops)
character(len=30)          :: fmt_csvoutput                      ! ----------- | (a fortran format spec for
     outputting csv style data)
real                       :: pipe_entering_temp                 ! [C] ------- | (the inlet temp of the pipe
      for a time step)
real                       :: segment_entering_temp              ! [C] ------- | (the temperature of fluid
     entering this segment)
real                       :: this_segment_temperature           ! [C] ------- | (the temperature of the
     current segment at the previous timestep)
real                       :: mixed_temperature                  ! [C] ------- | (for mixed flow, this is
     the mixed temperature, before any heat transfer calcs)
```

```
real                        :: current_time                         ! [s] ------- | (the current time of the
      simulation at the end of this time step)
real                        :: fluid_reynoldsnumber                 ! [-] ------- | (for heat transfer calcs,
      this represents the Reynolds number for current flow)
real                        :: fluid_nusseltNumber                  ! [-] ------- | (for heat transfer calcs,
      this represents the Nusselt number for current flow)
real                        :: fluid_convectioncoefficient          ! [W/m2-K] -- | (for heat transfer calcs,
      this represents the fluid to pipe inner wall convection coefficient)
real                        :: UA                                   ! [W/K] ----- | (for heat transfer calcs,
      this represents the conductance from fluid to pipe outer wall)
real                        :: resistance                           ! [m2-K/W] -- | (for heat transfer calcs,
      this represents the resistance per surf area from fluid to pipe outer wall)
real                        :: mass_coming_in_to_segment            ! [kg] ------ | (the amount of fluid mass
      that will be entering this segment from upstream)
real, dimension(4)          :: A                                    ! -varies---- | (coefficients of fluid heat
       balance equation)
real, dimension(4)          :: B                                    ! -varies---- | (coefficients of pipe heat
      balance equation)
real                        :: FluidNodeHeatCapacity                ! -???------- | (heat capacity of fluid
      portion of a segment)
real                        :: PipeHeatCapacity                     ! -???------- | (heat capacity of pipe
      portion of a segment)
real                        :: EnvHeatTransCoef                     ! -???------- | (outer convection
      coefficient of pipe...high value for type 1 boundary)
character(len=4)            :: s_index                              ! ----------- | (string placeholder for
      writing indeces)
integer                     :: time_step_counter                    ! ----------- | (simple counter for keeping
       track of time step index)
character(len=30)           :: s_inputs                             ! ----------- | (placeholder for a input
      file command line argument)
real                        :: expon                                ! [-] ------- | (Nusselt correlation
      exponent)
real, dimension(-3:-1)      :: model_temp                           ! [C] ------- | (stores the temperature
      calculated by each model type)

! spew
write(*,'(A)') "---␣Simulation␣starting!␣---"

! spew a useful output file using the default values -- only if we aren't already using one!
call get_command_argument(number=1, value=s_inputs)
if (len_trim(s_inputs)==0) then
    ! must not have had a CL argument
    call write_overridables(simData)
end if

! override with environment variables
call process_environment_variables(simData)

! override with input file -- this is more localized since the input file must be specified on the command line
call process_input_file(simData)

! read in boundary EFT data if applicable
if (simData%circtype == circtype_boundaryEFT) then
    call process_boundaryEFT_file()
end if

! read in boundary Q data if applicable
if (simData%heatpumptesttype == testtype_boundaryFile) then
    call process_boundaryHPQ_file()
end if

! allocate once we know the final number of segments (after overrides)
allocate(fluid_segments(simData%num_segments))
allocate(segmentTemps_prevTime(simData%num_segments))

! once input has been read and overridden, calculate the derived parameters
segment_length             = simData%total_pipe_length / simData%num_segments
pipe_cross_sectional_area  = (PI/4.0) * (simData%pipe_inner_diameter**2)
fluid_volume_flow_rate     = simData%fluid_mass_flow_rate / simData%fluid_density
fluid_velocity             = fluid_volume_flow_rate / pipe_cross_sectional_area
segment_residence_time     = segment_length / fluid_velocity
pipe_residence_time        = simData%total_pipe_length / fluid_velocity
time_step                  = pipe_residence_time / simData%num_segments
num_time_steps             = simData%max_time / time_step
segment_surface_area       = PI * simData%pipe_inner_diameter * segment_length
segment_out_surface_area   = PI * simData%pipe_outer_diameter * segment_length
fluid_mass_in_a_segment    = simData%fluid_density * pipe_cross_sectional_area * segment_length
time_heatpump_cutoff       = pipe_residence_time / 2.0

! initialize segment temperatures and names
fluid_segments%temperature = simData%initial_fluid_temp
do fluid_segment_index = 1, simData%num_segments
    write (s_index, '(I4)') fluid_segment_index
    fluid_segments(fluid_segment_index)%name = 'Segment'//trim(adjustl(s_index))
end do

! open the output file
open(file_csvfluidtemps, file='segment_temps.csv')

! write the header row format
write(fmt_csvoutput, '(a,i4,a)' ) '(', simData%num_segments+2, '(a12,","))'
write(file_csvfluidtemps, fmt_csvoutput) 'time', 'inlet␣temp', fluid_segments%name
```

```
! create the output format for data rows (re-use format variable)
write(fmt_csvoutput, '(a,i4,a)' ) '(', simData%num_segments+2, '(f10.4,","))'

! initialize the current time counter to 0 since it is at the beginning of the DO loop now
current_time = 0
time_step_counter = 0

! initialize the pipe entering temperature to the system here
if (simData%circtype == circtype_boundaryEFT) then
    pipe_entering_temp = get_boundaryEFT(current_time)
else
    pipe_entering_temp = simData%entering_fluid_temp
end if

! until further development, assign coefficients here:
if ( abs(sum(simData%model_coef) - 1.0) > 0.01 ) then
    call issuefatal('Mixing␣model␣coefficients␣do␣not␣sum␣to␣1')
end if

! loop over time until the current time reaches the maximum simulation time
do while (current_time <= simData%max_time)

    ! increment counter
    time_step_counter = time_step_counter + 1

    ! update current time for the next time step
    current_time = current_time + time_step

    ! store temperatures - keep in mind this stores the 'outlet' temperature of each segment, so the
         pipe_entering_temp isn't disrupted
    segmentTemps_prevTime = fluid_segments%temperature

    ! initialize the entering temp for the next time step
    if (simData%circtype == circtype_simpleRecirc) then
        ! don't add heat
        pipe_entering_temp = fluid_segments(simData%num_segments)%temperature
    elseif (simData%circtype == circtype_heatpump) then
        if (simData%heatpumptesttype == testtype_impulse .and. current_time > time_heatpump_cutoff) then
            ! don't add heat
            pipe_entering_temp = fluid_segments(simData%num_segments)%temperature
        else
            ! do add heat in all other cases (although heat added could be zero...)
            if (simData%heatpumptesttype == testtype_boundaryFile) then
                ! use the scheduled heat addition value to override the nominal entered rate
                simData%Q_heatpump = get_boundaryQ(current_time)
            end if
            pipe_entering_temp = fluid_segments(simData%num_segments)%temperature + simData%Q_heatpump / (simData
                 %fluid_mass_flow_rate * simData%fluid_specific_heat)
            if ((time_step_counter/simData%report_frequency) == (real(time_step_counter)/real(simData%
                 report_frequency)) .or. time_step_counter==num_time_steps .or. time_step_counter==1) then
                write(*,'("Heat␣added=",␣F8.3,␣";␣PipeOutletTemp=",␣F8.3,␣";␣NewPipeInletTemp=",␣F8.3)') simData%
                     Q_heatpump, fluid_segments(simData%num_segments)%temperature, pipe_entering_temp
            end if
        end if
    else if (simData%circtype == circtype_boundaryEFT) then
        ! the entering temp is scheduled
        pipe_entering_temp = get_boundaryEFT(current_time)
    end if

    ! calculate the convection coefficient if we are doing heat transfer calcs
    if (simData%heattransfertype /= heattransfertype_adiabatic) then
        fluid_reynoldsnumber = simData%pipe_inner_diameter * fluid_velocity / simData%fluid_kinematic_visc
        !if (pipe_outer_surface_temp >= pipe_entering_temp) then ! heating
            expon = 0.4
        !else ! cooling
        !    expon = 0.5
        !end if
        fluid_nusseltNumber = 0.023 * (fluid_reynoldsnumber ** 0.8) * (simData%fluid_prandtl ** expon)
        fluid_convectioncoefficient = fluid_nusseltNumber * simData%fluid_conductivity / simData%
             pipe_inner_diameter
        resistance = (1.0 / fluid_convectioncoefficient) + (log(simData%pipe_outer_diameter/simData%
             pipe_inner_diameter) / (2 * PI * simData%pipe_conductivity * segment_length))
        UA = (1 / resistance) * segment_surface_area
    end if

    ! loop over all fluid segements
    do fluid_segment_index = 1, simData%num_segments

        ! get segment entering temp
        segment_entering_temp = get_segment_entering_temp(segmentTemps_prevTime, fluid_segment_index,
             pipe_entering_temp)

        ! store this segment temperature
        this_segment_temperature = fluid_segments(fluid_segment_index)%temperature

        ! ******* Calculate plug flow temperature ******* !
        if (simData%heattransfertype == heattransfertype_adiabatic) then
            model_temp(modeltype_plugflow) = segment_entering_temp
        elseif (simData%heattransfertype == heattransfertype_pipeouterboundary) then
            model_temp(modeltype_plugflow) = segment_entering_temp + (UA / (fluid_mass_in_a_segment*simData%
                 fluid_specific_heat)) * (simData%pipe_outer_surface_temp - segment_entering_temp)
        end if
        ! ******* Calculate well-mixed temperature ******* !

        ! calculate incoming mass and mass already in the segment
        mass_coming_in_to_segment = simData%fluid_mass_flow_rate * time_step

        ! calculate a mixed temperature
        mixed_temperature = (mass_coming_in_to_segment * segment_entering_temp + fluid_mass_in_a_segment *
             this_segment_temperature) / (mass_coming_in_to_segment + fluid_mass_in_a_segment)

        ! do mixing calculation, either adiabatic or with heat transfer
        if (simData%heattransfertype == heattransfertype_adiabatic) then
            model_temp(modeltype_wellmixedsegments) = mixed_temperature
        elseif (simData%heattransfertype == heattransfertype_pipeouterboundary) then
            model_temp(modeltype_wellmixedsegments) = mixed_temperature + (UA / (fluid_mass_in_a_segment*simData%
                 fluid_specific_heat)) * (simData%pipe_outer_surface_temp - mixed_temperature)
        end if

        ! ******* Calculate Hanby temperature ******* !

        ! coef of fluid heat balance
        FluidNodeHeatCapacity = pipe_cross_sectional_area * segment_length * simData%fluid_specific_heat *
             simData%fluid_density ! Mass of Node x Specific heat
        A(1) = FluidNodeHeatCapacity + simData%fluid_mass_flow_rate * simData%fluid_specific_heat * time_step +
             fluid_convectioncoefficient * segment_surface_area * time_step
        A(2) = simData%fluid_mass_flow_rate * simData%fluid_specific_heat * time_step
        A(3) = fluid_convectioncoefficient * segment_surface_area * time_step
        A(4) = FluidNodeHeatCapacity

        ! coef of pipe heat balance
        PipeHeatCapacity = simData%pipe_specific_heat * simData%pipe_density * (pi * 0.25 * simData%
             pipe_outer_diameter**2 - pipe_cross_sectional_area)
        EnvHeatTransCoef = 100000.0 ! to simulate a temperature boundary
        B(1) = PipeHeatCapacity + fluid_convectioncoefficient * segment_surface_area * time_step +
             EnvHeatTransCoef * segment_out_surface_area * time_step
        B(2) = A(3)
        B(3) = EnvHeatTransCoef * segment_out_surface_area * time_step
        B(4) = PipeHeatCapacity

        ! use the Hanby expression (energy balance)
        model_temp(modeltype_hanby) = (A(2) * segment_entering_temp + A(3)/B(1) * (B(3)* simData%
             pipe_outer_surface_temp + B(4) * simData%pipe_outer_surface_temp) + A(4) * this_segment_temperature)
              /(A(1)-A(3)*B(2)/B(1))

        ! ******* Calculate final weighted temperature ******* !   (a dot product of coef and temp arrays)
        fluid_segments(fluid_segment_index)%temperature = sum(simData%model_coef * model_temp)

    end do

    ! flush output
    write(file_csvfluidtemps, fmt_csvoutput) current_time, pipe_entering_temp, fluid_segments%temperature

    ! report if desired
    if ((time_step_counter/simData%report_frequency) == (real(time_step_counter)/real(simData%report_frequency))
         .or. time_step_counter==num_time_steps .or. time_step_counter==1) then
        write(*,'("Just␣finished␣time␣step␣#",␣I8,␣"/",␣I8)') time_step_counter, num_time_steps
    end if

end do ! time step while loop

! we are done!
write(*,'(A)') "---␣Simulation␣complete!␣---"

! close the output file
close(file_csvfluidtemps)

! end program
end program
```

Listing C.2: Transport Delay Testbed Source: Data Structures

```
!This module contains data structures and minimal variables declared here
module structures

! use the enumeration definitions
use enumerations

! everything must be explicitly declared
implicit none

! everything is public here
public

! ================================================================= |
! = = = = = = = = = = = = = GLOBAL PARAMETERS = = = = = = = = = = = = = |
! ----------------------------------------------------------------- |
!~~~~~VAR TYPE~~~~~||~~~~~~~~VAR NAME~~~~~~~~~|~~VALUE~~~~~~~~~~~~~~~|
! ----------------------------------------------------------------- |
! -------------------------------OTHER------------------------------ |
real,     parameter :: PI                      = 3.14159  ! ----------- |
real,     parameter :: initVal                 = -99999.  ! ----------- |
! ----------------------------FILE UNIT NUMBERS--------------------- |
integer, parameter :: file_csvfluidtemps      = 28       ! ----------- |
integer, parameter :: file_boundaryEFT        = 30       ! ----------- |
integer, parameter :: file_inputs             = 32       ! ----------- |
integer, parameter :: file_overrides          = 34       ! ----------- |
integer, parameter :: file_boundaryQ          = 36       ! ----------- |
! -------------------------------FILE NAMES------------------------- |
character(len=15), parameter :: s_boundaryEFT = 'boundaryEFT.csv' ! - |
character(len=15), parameter :: s_boundaryQ   = 'boundaryQ.csv' ! --- |
! ================================================================= |

! = = = = = = = = = = = = = = = = = MAIN SIMULATION DATA STRUCTURE = = = = = = = = = = = = = = = = = |
type inputStruc
    real    :: initial_fluid_temp       = 16.577                        ! [C] ------ Approximated ------- |
! --------------------------BOUNDARY CONDITIONS-------------------------------------------------------- |
    real    :: entering_fluid_temp      = 16.577                        ! [C] ------ Approximated ------- |
    real    :: pipe_outer_surface_temp  = 16.577                        ! [C] ------ Approximated ------- |
    real    :: fluid_mass_flow_rate     = 0.29                          ! [kg/s] --- Experimental Setup - |
! --------------------------PIPE PARAMETERS------------------------------------------------------------ |
    real    :: pipe_inner_diameter      = 0.02154                       ! [m] ------ 3/4" HDPE SDR 11 --- |
    real    :: pipe_outer_diameter      = 0.02667                       ! [m] ------ 3/4" HDPE SDR 11 --- |
    real    :: total_pipe_length        = 121.92                        ! [m] ------ Experimental setup - |
    real    :: pipe_conductivity        = 0.45                          ! [W/m-K] -- EngineeringToolbox - |
    real    :: pipe_specific_heat       = 1950.0                        ! [J/kg-K] - Matt --------------- |
    real    :: pipe_density             = 950.0                         ! [kg/m3] -- Matt --------------- |
! --------------------------FLUID PARAMETERS----------------------------------------------------------- |
    real    :: fluid_conductivity       = 0.58                          ! [W/m-K] -- EngineeringToolbox - |
    real    :: fluid_density            = 997.8                         ! [kg/m3] -- EngineeringToolbox - |
    real    :: fluid_specific_heat      = 4187.0                        ! [J/kg-K] - EngineeringToolbox - |
    real    :: fluid_kinematic_visc     = 0.8e-6                        ! [m2/s] --- EngineeringToolbox - |
    real    :: fluid_prandtl            = 7.0                           ! [-] ------ EngineeringToolbox - |
! -----------------------HEAT PUMP PARAMETERS (if any)------------------------------------------------- |
    real    :: Q_heatpump               = 3500.0                        ! [W] ------ Experimental Setup - |
! --------------------------SIMULATION PARAMETERS------------------------------------------------------ |
    integer :: num_segments             = 20                            ! [-] ------ Model Parameter ---- |
    integer :: max_time                 = 1800                          ! [s] ------ Model Parameter ---- |
    integer :: report_frequency         = 100                           ! [timestep] Sim Parameter ------ |
! --------------------------MIXING MODEL CONTRIBUTIONS------------------------------------------------- |
    real    :: model_coef(-3:-1)        = (/1.0, 0.0, 0.0/)             ! [-] ------ Dim'd to model #s -- |
! --------------------------MODEL SETTINGS------------------------------------------------------------- |
    integer :: circtype                 = circtype_heatpump             ! [-] -------------------------- |
    integer :: heattransfertype         = heattransfertype_adiabatic    ! [-] -------------------------- |
    integer :: heatpumptesttype         = testtype_stepchange           ! [-] -------------------------- |
end type

! ============================================================== |
! = = = = = = = = = DEFINITION OF A SINGLE SEGMENT = = = = = = = = |
type fluid_segment ! ------------------------------------------- |
    ! temperature is just past the inlet of the segment ---------- |
    !  all the way to including the oulet of the segment --------- |
    real               :: temperature = initVal ! ----------------- |
    character(len=12) :: name          ='XXXXXX' ! ----------------- |
end type ! ---------------------------------------------------- |
! ============================================================== |

! ============================================================== |
! = = = = = = = DEFINITION OF A BOUNDARY POINT IN TIME = = = = = = |
type transientPoint ! ------------------------------------------ |
    real :: time_seconds = initVal ! ---------------------------- |
    real :: boundary_val = initVal ! ---------------------------- |
end type ! ---------------------------------------------------- |
! ============================================================== |

! ================================================================== |
! = = = = = = = = = ONLY A COUPLE INSTANCE VARIABLES = = = = = = = = = |
type(inputStruc) :: simData ! -------------------------------------- |
type(transientPoint), dimension(:), allocatable :: boundaryEFTdata !-- |
type(transientPoint), dimension(:), allocatable :: boundaryHPQdata !-- |
! ================================================================== |
```

```
end module
```

Listing C.3: Transport Delay Testbed Source: Enumerations

```
!This module contains enumerations and related utilities
module enumerations

! everything must be explicitly declared
implicit none

! everything is public here
public

! --------------VARIABLES-------------------------------------------- |
! -------------------------------MODEL TYPE ENUMERATION-------------- |
integer, parameter :: modeltype_plugflow      = -1       ! ----------- |
integer, parameter :: modeltype_wellmixedsegments = -2   ! ----------- |
integer, parameter :: modeltype_hanby         = -3       ! ----------- |
! -------------------------------CIRCULATION TYPE ENUMERATION-------- |
integer, parameter :: circtype_heatpump       = -1       ! ----------- |
integer, parameter :: circtype_simplerecirc   = -2       ! ----------- |
integer, parameter :: circtype_boundaryEFT    = -3       ! ----------- |
! -------------------------------HEAT TRANSFER TYPE ENUMERATION------ |
integer, parameter :: heattransfertype_adiabatic = -1    ! ----------- |
integer, parameter :: heattransfertype_pipeouterboundary = -2 ! ----- |
! -------------------------------TEST TYPE ENUMERATION--------------- |
integer, parameter :: testtype_impulse        = -1       ! ----------- |
integer, parameter :: testtype_stepchange     = -2       ! ----------- |
integer, parameter :: testtype_boundaryFile   = -3       ! ----------- |
! ------------------------------------------------------------------- |

! routine declarations
public get_circtype_int_from_string
public get_heattransfertype_int_from_string
public get_heatpumptesttype_int_from_string
public get_circtype_string_from_int
public get_heattransfertype_string_from_int
public get_heatpumptesttype_string_from_int

! actual routine code
contains

    integer function get_circtype_int_from_string(s) result(circtype)

        character(len=*), intent(in) :: s

        select case (trim(s))
        case ('HEATPUMP')
            circtype = circtype_heatpump
        case ('SIMPLERECIRC')
            circtype = circtype_simplerecirc
        case ('BOUNDARYEFT')
            circtype = circtype_boundaryEFT
        end select

    end function

    integer function get_heattransfertype_int_from_string(s) result(heattransfertype)

        character(len=*), intent(in) :: s

        select case (trim(s))
        case ('ADIABATIC')
            heattransfertype = heattransfertype_adiabatic
        case ('PIPEOUTERBOUNDARY')
            heattransfertype = heattransfertype_pipeouterboundary
        end select

    end function

    integer function get_heatpumptesttype_int_from_string(s) result(heatpumptesttype)

        character(len=*), intent(in) :: s

        select case (trim(s))
        case ('IMPULSE')
            heatpumptesttype = testtype_impulse
        case ('STEPCHANGE')
            heatpumptesttype = testtype_stepchange
        case ('BOUNDARYFILE')
            heatpumptesttype = testtype_boundaryFile
        end select

    end function

    character(len=20) function get_circtype_string_from_int(circtype) result(s)

        integer, intent(in) :: circtype

        select case (circtype)
        case (circtype_heatpump)
            s = 'HEATPUMP'
        case (circtype_simplerecirc)
```

```
            s = 'SIMPLERECIRC'
        case (circtype_boundaryEFT)
            s = 'BOUNDARYEFT'
        end select

    end function

    character(len=20) function get_heattransfertype_string_from_int(heattransfertype) result(s)

        integer, intent(in) :: heattransfertype

        select case (heattransfertype)
        case (heattransfertype_adiabatic)
            s = 'ADIABATIC'
        case (heattransfertype_pipeouterboundary)
            s = 'PIPEOUTERBOUNDARY'
        end select

    end function

    character(len=20) function get_heatpumptesttype_string_from_int(heatpumptesttype) result(s)

        integer, intent(in) :: heatpumptesttype

        select case (heatpumptesttype)
        case (testtype_impulse)
            s = 'IMPULSE'
        case (testtype_stepchange)
            s = 'STEPCHANGE'
        case (testtype_boundaryFile)
            s = 'BOUNDARYFILE'
        end select

    end function

end module
```

Listing C.4: Transport Delay Testbed Source: Utilities

```fortran
module transportdelay_utilities

! allow use of data structures and global variables
use structures

! use the enumeration definitions
use enumerations

! explicitly declare everything!
implicit none

! all things private unless explicitly public
private

interface process_environment_variable
    module procedure process_environment_variable_f
    module procedure process_environment_variable_i
end interface

! scope specifications of routines in this module
public  issuefatal
public  get_segment_entering_temp
public  process_environment_variables
public  process_boundaryEFT_file
public  process_boundaryHPQ_file
public  process_input_file
public  write_overridables
private to_upper
public  get_boundaryEFT
public  get_boundaryQ

! actual routine code
contains

    subroutine issuefatal(s)

        character(len=*), intent(in) :: s

        write(*,*) '***********************'
        write(*,*) '******FATAL␣ERROR******'
        write(*,*) '***********************'
        write(*,*) s
        write(*,*) '***********************'
        write(*,*) 'press␣ENTER␣to␣exit...'
        read(*,*)
        call exit(1)

    end subroutine

    real function get_segment_entering_temp(fluid_segments, fluid_segment_index, pipe_entering_temp) result(
         segment_entering_temp)

        real, dimension(*), intent(in) :: fluid_segments
        integer, intent(in) :: fluid_segment_index
        real, intent(in) :: pipe_entering_temp

        !get a fluid temperature for this one
        if (fluid_segment_index==1) then
          segment_entering_temp = pipe_entering_temp
        else
          segment_entering_temp = fluid_segments(fluid_segment_index-1)
        end if

    end function

    subroutine process_environment_variables(inputs)

        type(inputStruc), intent(in out) :: inputs

        character(len=40)          :: tmp_env_variable_value                   ! --------- | (holds the string
             retrieved from the env var)
        integer :: var_stat

        call process_environment_variable('MODELCOEFHANBY', 'Hanby␣model␣mixing␣coefficient', inputs%model_coef(
             modeltype_hanby))
        call process_environment_variable('MODELCOEFPLUGFLOW', 'Plug␣flow␣model␣mixing␣coefficient', inputs%
             model_coef(modeltype_plugflow))
        call process_environment_variable('MODELCOEFWELLMIXED', 'Well-mixed␣model␣mixing␣coefficient', inputs%
             model_coef(modeltype_wellmixedsegments))

        call get_environment_variable('CIRCTYPE', tmp_env_variable_value)
        select case (trim(tmp_env_variable_value))
        case ('HEATPUMP')
            inputs%circtype = circtype_heatpump
            write(*,'(A)') "Environment␣Variable:␣Overrode␣circ␣type␣with␣HEATPUMP"
        case ('SIMPLERECIRC')
            inputs%circtype = circtype_simplerecirc
            write(*,'(A)') "Environment␣Variable:␣Overrode␣circ␣type␣with␣SIMPLERECIRC"
        case ('BOUNDARYEFT')
            inputs%circtype = circtype_boundaryEFT
            write(*,'(A)') "Environment␣Variable:␣Overrode␣circ␣type␣with␣BOUNDARYEFT"
        end select

        call get_environment_variable('HEATTRANSFERTYPE', tmp_env_variable_value)
        select case (trim(tmp_env_variable_value))
        case ('ADIABATIC')
            inputs%heattransfertype = heattransfertype_adiabatic
            write(*,'(A)') "Environment␣Variable:␣Overrode␣HT␣type␣with␣ADIABATIC"
        case ('PIPEOUTERBOUNDARY')
            inputs%heattransfertype = heattransfertype_pipeouterboundary
            write(*,'(A)') "Environment␣Variable:␣Overrode␣HT␣type␣with␣PIPEOUTERBOUNDARY"
        end select

        call get_environment_variable('TESTTYPE', tmp_env_variable_value)
        select case (trim(tmp_env_variable_value))
        case ('IMPULSE')
            inputs%heatpumptesttype = testtype_impulse
            write(*,'(A)') "Environment␣Variable:␣Overrode␣test␣type␣with␣IMPULSE"
        case ('STEPCHANGE')
            inputs%heatpumptesttype = testtype_stepchange
            write(*,'(A)') "Environment␣Variable:␣Overrode␣test␣type␣with␣STEPCHANGE"
        case ('BOUNDARYFILE')
            inputs%heatpumptesttype = testtype_boundaryFile
            write(*,'(A)') "Environment␣Variable:␣Overrode␣test␣type␣with␣BOUNDARYFILE"
        end select

        call process_environment_variable('INITIAL_FLUID_TEMP', 'init␣fluid␣temp', inputs%initial_fluid_temp)
        call process_environment_variable('ENTERING_FLUID_TEMP', 'entering␣fluid␣temp', inputs%
             entering_fluid_temp)
        call process_environment_variable('PIPE_OUTER_SURFACE_TEMP', 'pipe␣outer␣surface␣temp', inputs%
             pipe_outer_surface_temp)
        call process_environment_variable('FLUID_MASS_FLOW_RATE', 'fluid␣mass␣flow␣rate', inputs%
             fluid_mass_flow_rate)
        call process_environment_variable('PIPE_INNER_DIAMETER', 'pipe␣inner␣diameter', inputs%
             pipe_inner_diameter)
        call process_environment_variable('PIPE_OUTER_DIAMETER', 'pipe␣outer␣diameter', inputs%
             pipe_outer_diameter)
        call process_environment_variable('TOTAL_PIPE_LENGTH', 'total␣pipe␣length', inputs%total_pipe_length)
        call process_environment_variable('PIPE_CONDUCTIVITY', 'pipe␣conductivity', inputs%pipe_conductivity)
        call process_environment_variable('PIPE_SPECIFIC_HEAT', 'pipe␣specific␣heat', inputs%pipe_specific_heat)
        call process_environment_variable('PIPE_DENSITY', 'pipe␣density', inputs%pipe_density)
        call process_environment_variable('FLUID_CONDUCTIVITY', 'fluid␣conductivity', inputs%fluid_conductivity)
        call process_environment_variable('FLUID_DENSITY', 'fluid␣density', inputs%fluid_density)
        call process_environment_variable('FLUID_SPECIFIC_HEAT', 'fluid␣specific␣heat', inputs%
             fluid_specific_heat)
        call process_environment_variable('FLUID_KINEMATIC_VISC', 'fluid␣kinematic␣viscosity', inputs%
             fluid_kinematic_visc)
        call process_environment_variable('FLUID_PRANDTL', 'fluid␣prandtl', inputs%fluid_prandtl)
        call process_environment_variable('Q_HEATPUMP', 'heat␣addition', inputs%Q_heatpump)
        call process_environment_variable('NUM_SEGMENTS', 'number␣of␣segments', inputs%num_segments)
        call process_environment_variable('MAX_TIME', 'max␣simulation␣time', inputs%max_time)
        call process_environment_variable('REPORT_FREQUENCY', 'reporting␣frequency', inputs%report_frequency)

    end subroutine

    subroutine process_environment_variable_f(varKey, varName, var)

        character(len=*), intent(in) :: varKey
        character(len=*), intent(in) :: varName
        real, intent(inOut) :: var

        integer :: var_stat
        character(len=40)          :: tmp_env_variable_value                  ! --------- | (holds the string
             retrieved from the env var)

        call get_environment_variable(varKey, tmp_env_variable_value, status=var_stat)
        if (var_stat == 0) then
            read(tmp_env_variable_value, *) var
            write(*,'("Environment␣Variable:␣Overrode␣",␣A20,␣"␣=␣",␣F8.3)') varName, var
        end if

    end subroutine

    subroutine process_environment_variable_i(varKey, varName, var)

        character(len=*), intent(in) :: varKey
        character(len=*), intent(in) :: varName
        integer, intent(inOut) :: var

        integer :: var_stat
        character(len=40)          :: tmp_env_variable_value                  ! --------- | (holds the string
             retrieved from the env var)

        call get_environment_variable(varKey, tmp_env_variable_value, status=var_stat)
        if (var_stat == 0) then
            read(tmp_env_variable_value, *) var
            write(*,'("Environment␣Variable:␣Overrode␣",␣A20,␣"␣=␣",␣I12)') varName, var
        end if

    end subroutine

    subroutine process_boundaryEFT_file()
```

```
        ! we will assume the file form is of the following:
        ! row 1: header
        ! rows 2-N: timestamp,temperature
        ! timestamp is in seconds, temperature is in celsius
        ! max line length = 200

        integer, parameter :: maxrows = 1000000

        integer :: Comma
        integer :: LineLength
        character(len=200) ReadLine
        integer :: IOStatus
        integer :: DataPointCtr
        type(transientPoint), dimension(:), allocatable :: TEMP_boundaryEFTdata

        ! allocate the temp array
        allocate(TEMP_boundaryEFTdata(maxrows))

        ! open the file for reading
        open(file_boundaryEFT, file=s_boundaryEFT, err=2980)

        ! init the counter
        DataPointCtr = -1

        ! start reading, line by line
        do

            ! read the line
            read(file_boundaryEFT, '(A)', iostat=IOStatus) ReadLine

            ! do counter things
            DataPointCtr = DataPointCtr + 1
            if (DataPointCtr == 0) cycle
            if (DataPointCtr >= maxrows) exit ! something is wrong!

            ! check for errors
            if (IOStatus .NE. 0) exit

            ! if this is a blank line, we are done here
            if (len_trim(adjustl(ReadLine)) == 0) exit

            !Read TimeStamp
            ReadLine = ADJUSTL(ReadLine)
            Comma = SCAN(ReadLine, ',')
            read(ReadLine(1:Comma-1), *) TEMP_boundaryEFTdata(DataPointCtr)%time_seconds
            LineLength = len(ReadLine)
            ReadLine = ReadLine(Comma+1:LineLength)

            !Read EFT
            ReadLine = ADJUSTL(ReadLine)
            Comma = SCAN(ReadLine, ',')
            if (Comma /= 0) then
                ReadLine = ReadLine(1:Comma-1)
            end if
            read(ReadLine, *) TEMP_boundaryEFTdata(DataPointCtr)%boundary_val

        end do

        ! mop up duty
        close(file_boundaryEFT)

        ! now allocate and assign the actual array
        allocate(boundaryEFTdata(DataPointCtr-1))
        boundaryEFTdata = TEMP_boundaryEFTdata(1:DataPointCtr-1)

        ! more mop up
        deallocate(TEMP_boundaryEFTdata)

        ! return early to bypass the error handler
        return

!ErrHandler
2980 CONTINUE
        WRITE(*, '(A)') 'Could␣not␣find␣'//s_boundaryEFT//'..aborting...'
        STOP

    end subroutine

    subroutine process_boundaryHPQ_file()

        ! we will assume the file form is of the following:
        ! row 1: header
        ! rows 2-N: timestamp,heatAdditionToLoop
        ! timestamp is in seconds, heatAdditionToLoop is in watts
        ! max line length = 200

        integer, parameter :: maxrows = 1000000

        integer :: Comma
        integer :: LineLength
        character(len=200) ReadLine
        integer :: IOStatus
        integer :: DataPointCtr
        type(transientPoint), dimension(:), allocatable :: TEMP_boundaryQdata

        ! allocate the temp array
        allocate(TEMP_boundaryQdata(maxrows))

        ! open the file for reading
        open(file_boundaryQ, file=s_boundaryQ, err=2980)

        ! init the counter
        DataPointCtr = -1

        ! start reading, line by line
        do

            ! read the line
            read(file_boundaryQ, '(A)', iostat=IOStatus) ReadLine

            ! do counter things
            DataPointCtr = DataPointCtr + 1
            if (DataPointCtr == 0) cycle
            if (DataPointCtr >= maxrows) exit ! something is wrong!

            ! check for errors
            if (IOStatus .NE. 0) exit

            ! if this is a blank line, we are done here
            if (len_trim(adjustl(ReadLine)) == 0) exit

            !Read TimeStamp
            ReadLine = ADJUSTL(ReadLine)
            Comma = SCAN(ReadLine, ',')
            read(ReadLine(1:Comma-1), *) TEMP_boundaryQdata(DataPointCtr)%time_seconds
            LineLength = len(ReadLine)
            ReadLine = ReadLine(Comma+1:LineLength)

            !Read EFT
            ReadLine = ADJUSTL(ReadLine)
            Comma = SCAN(ReadLine, ',')
            if (Comma /= 0) then
                ReadLine = ReadLine(1:Comma-1)
            end if
            read(ReadLine, *) TEMP_boundaryQdata(DataPointCtr)%boundary_val

        end do

        ! mop up duty
        close(file_boundaryQ)

        ! now allocate and assign the actual array
        allocate(boundaryHPQdata(DataPointCtr-1))
        boundaryHPQdata = TEMP_boundaryQdata(1:DataPointCtr-1)

        ! more mop up
        deallocate(TEMP_boundaryQdata)

        ! return early to bypass the error handler
        return

!ErrHandler
2980 CONTINUE
        WRITE(*, '(A)') 'Could␣not␣find␣'//s_boundaryEFT//'..aborting...'
        STOP

    end subroutine

    subroutine process_input_file(inputs)

        type(inputStruc), intent(in out) :: inputs

        ! we will assume the file form is of the following:
        ! ! possible comments following exclamation points
        ! <blank lines are ignored>
        ! key = value
        ! key=value
        ! key =value !trailing comment
        ! max line length = 200

        integer :: Equals
        integer :: Exclamation
        character(len=200) ReadLine
        integer :: IOStatus
        integer :: LineCtr
        character(len=200) :: sKey
        character(len=200) :: sValue
        logical :: isthere
        integer :: errStatus
        character(len=30) :: s_inputs

        call get_command_argument(number=1, value=s_inputs, status=errStatus)
        if (len_trim(s_inputs)==0) then
            ! must not have had a CL argument
            return
```

```
else if (errStatus /= 0) then
    ! something went wrong, either s_inputs couldn't hold the string or something
    write(*,*) 'problem␣with␣CL␣processing'
end if

write(*,'(A)') 'Command␣line␣argument␣(input␣file␣name)␣found:␣'//trim(adjustl(s_inputs))

inquire(file=s_inputs, exist=isthere)
if (.not. isthere) then
    write(*,'(A)') '␣*␣Could␣not␣retrieve␣input␣file...continuing␣without␣input␣file␣modifications!'
    return
end if

! open the file for reading
open(file_inputs, file=s_inputs)

! init the counter
LineCtr = 0

! start reading, line by line
do

    ! do counter things
    LineCtr = LineCtr + 1

    ! read the line
    read(file_inputs, '(A)', iostat=IOStatus) ReadLine

    ! check for errors
    if (IOStatus .NE. 0) exit

    ! if this is a blank line, cycle
    if (len_trim(adjustl(ReadLine)) == 0) cycle

    ! remove leading spaces
    ReadLine = trim(adjustl(ReadLine))

    ! check for exclamation, take everything left of it and adjustl/trim it
    Exclamation = scan(ReadLine, '!')
    if (Exclamation == 1) then
        cycle !the first non-whitespace was an exclamation
    else if (Exclamation /= 0) then
        ReadLine = trim(adjustl(ReadLIne(1:Exclamation-1)))
    end if

    !Read Key
    Equals = SCAN(ReadLine, '=')
    READ(ReadLine(1:Equals-1), '(A)') sKey
    call to_upper(sKey)
    ReadLine = ReadLine(Equals+1:)

    !Read Val
    ReadLine = ADJUSTL(ReadLine)
    READ(ReadLine, '(A)') sValue
    call to_upper(sValue)

    ! process it
    select case (trim(adjustl(sKey)))
    case('MODELCOEFHANBY')
        read(sValue, *) inputs%model_coef(modeltype_hanby)
        write(*,'("Input␣File:␣Overrode␣Hanby␣model␣mixing␣coefficient␣=␣",␣F8.3)') inputs%model_coef(
             modeltype_hanby)
    case('MODELCOEFPLUGFLOW')
        read(sValue, *) inputs%model_coef(modeltype_plugflow)
        write(*,'("Input␣File:␣Overrode␣Plug␣flow␣model␣mixing␣coefficient␣=␣",␣F8.3)') inputs%model_coef
             (modeltype_plugflow)
    case('MODELCOEFWELLMIXED')
        read(sValue, *) inputs%model_coef(modeltype_wellmixedsegments)
        write(*,'("Input␣File:␣Overrode␣Hanby␣model␣mixing␣coefficient␣=␣",␣F8.3)') inputs%model_coef(
             modeltype_wellmixedsegments)
    case ('CIRCTYPE')
        select case (trim(adjustl(sValue)))
        case ('HEATPUMP')
            inputs%circtype = circtype_heatpump
        case ('SIMPLERECIRC')
            inputs%circtype = circtype_simplerecirc
        case ('BOUNDARYEFT')
            inputs%circtype = circtype_boundaryEFT
        case default
            call issuefatal('Invalid␣value␣for␣input␣variable␣CIRCTYPE:␣'//trim(adjustl(sValue)))
        end select
        write(*,'(A)') 'Input␣File:␣Overrode␣circ␣type␣with:␣'//trim(adjustl(sValue))
    case ('HEATTRANSFERTYPE')
        select case (trim(adjustl(sValue)))
        case ('ADIABATIC')
            inputs%heattransfertype = heattransfertype_adiabatic
        case ('PIPEOUTERBOUNDARY')
            inputs%heattransfertype = heattransfertype_pipeouterboundary
        case default
            call issuefatal('Invalid␣value␣for␣input␣variable␣HEATTRANSFERTYPE:␣'//trim(adjustl(sValue)))
        end select
        write(*,'(A)') 'Input␣File:␣Overrode␣heat␣transfer␣type␣with:␣'//trim(adjustl(sValue))
    case ('HEATPUMPTESTTYPE')
```

```
            select case (trim(adjustl(sValue)))
            case ('IMPULSE')
                inputs%heatpumptesttype = testtype_impulse
            case ('STEPCHANGE')
                inputs%heatpumptesttype = testtype_stepchange
            case ('BOUNDARYFILE')
                inputs%heatpumptesttype = testtype_boundaryFile
            case default
                call issuefatal('Invalid␣value␣for␣input␣variable␣HEATPUMPTESTTYPE:␣'//trim(adjustl(sValue)))
            end select
            write(*,'(A)') 'Input␣File:␣Overrode␣test␣type␣with:␣'//trim(adjustl(sValue))
        case ('INITIAL_FLUID_TEMP')
            read(sValue, *) inputs%initial_fluid_temp
            write(*,'("Input␣File:␣Overrode␣init␣fluid␣temp␣=␣",␣F8.3)') inputs%initial_fluid_temp
        case ('ENTERING_FLUID_TEMP')
            read(sValue, *) inputs%entering_fluid_temp
            write(*,'("Input␣File:␣Overrode␣entering␣fluid␣temp␣=␣",␣F8.3)') inputs%entering_fluid_temp
        case ('PIPE_OUTER_SURFACE_TEMP')
            read(sValue, *) inputs%pipe_outer_surface_temp
            write(*,'("Input␣File:␣Overrode␣pipe␣outer␣surface␣temp␣=␣",␣F8.3)') inputs%
                 pipe_outer_surface_temp
        case ('FLUID_MASS_FLOW_RATE')
            read(sValue, *) inputs%fluid_mass_flow_rate
            write(*,'("Input␣File:␣Overrode␣fluid␣mass␣flow␣rate␣=␣",␣F8.3)') inputs%fluid_mass_flow_rate
        case ('PIPE_INNER_DIAMETER')
            read(sValue, *) inputs%pipe_inner_diameter
            write(*,'("Input␣File:␣Overrode␣pipe␣inner␣diameter␣=␣",␣F8.3)') inputs%pipe_inner_diameter
        case ('PIPE_OUTER_DIAMETER')
            read(sValue, *) inputs%pipe_outer_diameter
            write(*,'("Input␣File:␣Overrode␣pipe␣outer␣diameter␣=␣",␣F8.3)') inputs%pipe_outer_diameter
        case ('TOTAL_PIPE_LENGTH')
            read(sValue, *) inputs%total_pipe_length
            write(*,'("Input␣File:␣Overrode␣total␣pipe␣length␣=␣",␣F8.3)') inputs%total_pipe_length
        case ('PIPE_CONDUCTIVITY')
            read(sValue, *) inputs%pipe_conductivity
            write(*,'("Input␣File:␣Overrode␣pipe␣conductivity␣=␣",␣F8.3)') inputs%pipe_conductivity
        case ('PIPE_SPECIFIC_HEAT')
            read(sValue, *) inputs%pipe_specific_heat
            write(*,'("Input␣File:␣Overrode␣pipe␣specific␣heat␣=␣",␣F8.3)') inputs%pipe_specific_heat
        case ('PIPE_DENSITY')
            read(sValue, *) inputs%pipe_density
            write(*,'("Input␣File:␣Overrode␣pipe␣density␣=␣",␣F8.3)') inputs%pipe_density
        case ('FLUID_CONDUCTIVITY')
            read(sValue, *) inputs%fluid_conductivity
            write(*,'("Input␣File:␣Overrode␣fluid␣conductivity␣=␣",␣F8.3)') inputs%fluid_conductivity
        case ('FLUID_DENSITY')
            read(sValue, *) inputs%fluid_density
            write(*,'("Input␣File:␣Overrode␣fluid␣density␣=␣",␣F8.3)') inputs%fluid_density
        case ('FLUID_SPECIFIC_HEAT')
            read(sValue, *) inputs%fluid_specific_heat
            write(*,'("Input␣File:␣Overrode␣fluid␣specific␣heat␣=␣",␣F8.3)') inputs%fluid_specific_heat
        case ('FLUID_KINEMATIC_VISC')
            read(sValue, *) inputs%fluid_kinematic_visc
            write(*,'("Input␣File:␣Overrode␣fluid␣kinematic␣viscosity␣=␣",␣F8.3)') inputs%
                 fluid_kinematic_visc
        case ('FLUID_PRANDTL')
            read(sValue, *) inputs%fluid_prandtl
            write(*,'("Input␣File:␣Overrode␣fluid␣prandtl␣number␣=␣",␣F8.3)') inputs%fluid_prandtl
        case ('Q_HEATPUMP')
            read(sValue, *) inputs%Q_heatpump
            write(*,'("Input␣File:␣Overrode␣heat␣addition␣=␣",␣F8.3)') inputs%Q_heatpump
        case ('NUM_SEGMENTS')
            read(sValue, *) inputs%num_segments
            write(*,'("Input␣File:␣Overrode␣number␣of␣segments␣=␣",␣I12)') inputs%num_segments
        case ('MAX_TIME')
            read(sValue, *) inputs%max_time
            write(*,'("Input␣File:␣Overrode␣max␣simulation␣time␣=␣",␣I12)') inputs%max_time
        case ('REPORT_FREQUENCY')
            read(sValue, *) inputs%report_frequency
            write(*,'("Input␣File:␣Overrode␣reporting␣frequency␣=␣",␣I12)') inputs%report_frequency
        case default
            call issuefatal('Invalid␣input␣variable␣found␣in␣the␣input␣file:␣'//trim(sValue))
        end select

    end do

    ! mop up duty
    close(file_inputs)

end subroutine

subroutine write_overridables(simData)

    type(inputStruc), intent(in) :: simData

    character(len=100)    :: tmpVar ! - temporary placeholder
    integer, dimension(8) :: vals   ! - holds date/time info
    integer               :: i      ! - simple counter

    open(file_overrides, file='overrides.txt')

    call date_and_time(values=vals)
```

```
write(file_overrides, '("!␣Created␣at:␣",␣I4,␣"-",␣I2,␣"-",␣I2,␣"␣␣",␣I2,␣":",␣I2,␣":",␣I2)') vals(1),
     vals(2), vals(3), vals(5), vals(6), vals(7)

write(file_overrides, '("!This␣file␣contains␣all␣the␣keys␣that␣can␣be␣overridden␣in␣the␣program.")')
write(file_overrides, '("!They␣may␣be␣overridden␣using␣environment␣variables␣at␣the␣command␣line,␣using␣
     CAPITALs␣for␣the␣key.")')
write(file_overrides, '("!They␣may␣also␣be␣overridden␣using␣an␣input␣file␣approach.")')
write(file_overrides, '("!The␣input␣file␣name␣(path)␣is␣optionally␣passed␣to␣the␣program␣as␣the␣only␣
     command␣line␣argument.")')
write(file_overrides, '("!The␣input␣file␣format␣is␣flexible,␣using␣a␣KEY=value␣syntax,␣''!''␣characters␣
     allow␣trailing␣comments,␣and␣whitespace␣(except␣for␣newlines)␣is␣ignored.")')
write(file_overrides, '("!All␣of␣these␣variables␣have␣default␣parameters,␣so␣any␣may␣be␣overridden,␣but␣
     none␣have␣to␣be.")')
write(file_overrides, '("!In␣this␣file,␣the␣defaults␣are␣listed␣here,␣thus␣this␣file␣may␣be␣used␣directly
     ␣as␣a␣*default*␣input␣file.")')

write(file_overrides, '␣')
write(file_overrides, '("!␣+␣+␣+␣Begin␣Variables␣+␣+␣+␣!")')

write(file_overrides, '␣')
write(file_overrides, '("␣␣!!!␣Physical␣properties␣!!!␣␣")')

write(tmpVar, *) simData%initial_fluid_temp
write(file_overrides,'("INITIAL_FLUID_TEMP=",A)') trim(adjustl(tmpVar))

write(tmpVar, *) simData%entering_fluid_temp
write(file_overrides,'("ENTERING_FLUID_TEMP=",A,"␣␣␣␣␣!-␣except␣in␣special␣cases,␣this␣should␣=␣
     INITIAL_FLUID_TEMP")') trim(adjustl(tmpVar))
write(tmpVar, *) simData%pipe_outer_surface_temp
write(file_overrides,'("PIPE_OUTER_SURFACE_TEMP=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%fluid_mass_flow_rate
write(file_overrides,'("FLUID_MASS_FLOW_RATE=",A)') trim(adjustl(tmpVar))

write(tmpVar, *) simData%pipe_inner_diameter
write(file_overrides,'("PIPE_INNER_DIAMETER=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%pipe_outer_diameter
write(file_overrides,'("PIPE_OUTER_DIAMETER=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%total_pipe_length
write(file_overrides,'("TOTAL_PIPE_LENGTH=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%pipe_conductivity
write(file_overrides,'("PIPE_CONDUCTIVITY=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%pipe_specific_heat
write(file_overrides,'("PIPE_SPECIFIC_HEAT=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%pipe_density
write(file_overrides,'("PIPE_DENSITY=",A)') trim(adjustl(tmpVar))

write(tmpVar, *) simData%fluid_conductivity
write(file_overrides,'("FLUID_CONDUCTIVITY=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%fluid_density
write(file_overrides,'("FLUID_DENSITY=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%fluid_specific_heat
write(file_overrides,'("FLUID_SPECIFIC_HEAT=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%fluid_kinematic_visc
write(file_overrides,'("FLUID_KINEMATIC_VISC=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%fluid_prandtl
write(file_overrides,'("FLUID_PRANDTL=",A)') trim(adjustl(tmpVar))

write(tmpVar, *) simData%Q_heatpump
write(file_overrides,'("Q_HEATPUMP=",A,"␣␣␣␣␣!-␣ignored␣if␣HEATPUMPTESTTYPE=BOUNDARYFILE")') trim(adjustl
     (tmpVar))

write(file_overrides, '␣')
write(file_overrides, '("␣␣!!!␣Simulation␣properties␣!!!␣␣")')

write(tmpVar, *) simData%num_segments
write(file_overrides,'("NUM_SEGMENTS=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%max_time
write(file_overrides,'("MAX_TIME=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%report_frequency
write(file_overrides,'("REPORT_FREQUENCY=",A)') trim(adjustl(tmpVar))

write(file_overrides, '␣')
write(file_overrides, '("␣␣!!!␣MIXING␣MODEL␣CONTRIBUTIONS␣-␣MUST␣SUM␣TO␣1.00  0 .01␣!!!␣␣")')

write(tmpVar, *) simData%model_coef(modeltype_hanby)
write(file_overrides,'("MODELCOEFHANBY=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%model_coef(modeltype_plugflow)
write(file_overrides,'("MODELCOEFPLUGFLOW=",A)') trim(adjustl(tmpVar))
write(tmpVar, *) simData%model_coef(modeltype_wellmixedsegments)
write(file_overrides,'("MODELCOEFWELLMIXED=",A)') trim(adjustl(tmpVar))

write(file_overrides, '␣')
write(file_overrides, '("␣␣!!!␣LOOP␣CIRCULATION␣!!!␣␣")')

write(file_overrides,'("CIRCTYPE=",A)') trim(adjustl(get_circtype_string_from_int(simData%circtype)))
do i = -3, -1
    if (i /= simData%circtype) write(file_overrides,'("!-CIRCTYPE=",A)') trim(adjustl(
         get_circtype_string_from_int(i)))
end do

write(file_overrides, '␣')
write(file_overrides, '("␣␣!!!␣HEAT␣TRANSFER␣MODEL␣!!!␣␣")')
```

```
        write(file_overrides,'("HEATTRANSFERTYPE=",A)') trim(adjustl(get_heattransfertype_string_from_int(simData
             %heattransfertype)))
        do i = -2, -1
            if (i /= simData%heattransfertype) write(file_overrides,'("!-HEATTRANSFERTYPE=",A)') trim(adjustl(
                 get_heattransfertype_string_from_int(i)))
        end do

        write(file_overrides, '␣')
        write(file_overrides, '("␣␣!!!␣HEAT␣PUMP␣TEST␣APPROACH␣!!!␣␣")')

        write(file_overrides,'("HEATPUMPTESTTYPE=",A)') trim(adjustl(get_heatpumptesttype_string_from_int(simData
             %heatpumptesttype)))
        do i = -3, -1
            if (i /= simData%heatpumptesttype) write(file_overrides,'("!-HEATPUMPTESTTYPE=",A)') trim(adjustl(
                 get_heatpumptesttype_string_from_int(i)))
        end do

        write(file_overrides, '␣')
        write(file_overrides, '("!␣+␣+␣+␣End␣Variables␣+␣+␣+␣!")')

        close(file_overrides)

    end subroutine

    subroutine to_upper(str)

        character(*), intent(in out) :: str
        integer :: i

        do i = 1, len(str)
            select case(str(i:i))
            case("a":"z")
                str(i:i) = ACHAR(IACHAR(str(i:i))-32)
            end select
        end do

    end subroutine To_upper

    real function get_boundaryEFT(current_time) result(EFT)

        real, intent(in) :: current_time

        integer :: i

        ! initialize to the first index
        EFT = boundaryEFTdata(1)%boundary_val

        ! now loop over the array, and if we are at that timestamp or above, take the new time
        do i = 1, size(boundaryEFTdata)
            if (current_time >= boundaryEFTdata(i)%time_seconds) then
                EFT = boundaryEFTdata(i)%boundary_val
            end if
        end do

    end function

    real function get_boundaryQ(current_time) result(Q)

        real, intent(in) :: current_time

        integer :: i

        ! initialize to the first index
        Q = boundaryHPQdata(1)%boundary_val

        ! now loop over the array, and if we are at that timestamp or above, take the new time
        do i = 1, size(boundaryHPQdata)
            if (current_time >= boundaryHPQdata(i)%time_seconds) then
                Q = boundaryHPQdata(i)%boundary_val
            end if
        end do

    end function

end module
```

VITA

Edwin Lee

Candidate for the Degree of

Doctor of Philosophy

Dissertation: AN IMPROVED HYDRONIC LOOP SYSTEM SOLUTION ALGORITHM WITH A ZONE-COUPLED HORIZONTAL GROUND HEAT EXCHANGER MODEL FOR WHOLE BUILDING ENERGY SIMULATION

Major Field: Mechanical Engineering

Biographical:

Personal Data: Born in Tulsa, Oklahoma, United States of America on October 29, 1982.

Education:
Received the B.S. degree from Oklahoma State University, Stillwater, Oklahoma, 2006, in Mechanical Engineering
Received the M.S. degree from Oklahoma State University, Stillwater, Oklahoma, 2008, in Mechanical Engineering

Experience:
Worked as an engineering intern with Specific Systems, LTD. in Tulsa, OK during the Summer of 2005 analyzing the design of custom HVAC equipment. Developed and implemented a buried pipe heat transfer model for EnergyPlus. Worked as a consultant for Oak Ridge National Laboratory performing EnergyPlus simulations of different building envelope materials, including phase change materials. Worked as a member of the core EnergyPlus development team developing new models, performing code repair and performing simulations.

Professional Memberships:
ASHRAE student member 2005-Present
President of ASHRAE student branch 2007-2012