Introduction

Much attention is paid in both college curriculum and classroom writing to the functioning of nuclear reactors, but equally important to the kinetics within the reactor itself is the operation of the balance of plant as a whole. These operations are often what determines the overall thermodynamic efficiency of the entire plant, and the control systems present within it can have as much of an effect on the conditions within the reactor as the behavior of the neutrons of the thermo-hydraulic relationship between the reactor moderator and overall reactivity. Tripped turbines can result in pressure increases within the reactor and are an essential part of most startup sequences (especially for Supercritical Water Reactors)[[1]](#footnote-1), poor control of condensers can lead to cavitation in pumps and loss of coolant flow to the reactors (which interestingly enough is one of the proposed causes of the Chernobyl Disaster)[[2]](#footnote-2), and pressure suppression systems can result in instability if not managed or modelled properly (another proposed cause of the Chernobyl Disaster)[[3]](#footnote-3). It is therefore imperative that reactor kinetics be simulated in concert with the behavior of the system designed to support and draw power from it. This is however quite difficult, as simulating the reactor itself is already a daunting enough task. As of yet, there are very few purpose-built systems for simulating the behavior of an entire nuclear plant, with most of these being training programs designed to model only a single specific nuclear plant.

The reason for the lack of large amounts of plant simulation software is probably due to the complexity of simulating so many different systems at once (the reactor, pumps, condensers, turbines and control systems, all with dozens of differential equations required to model them properly). However, a solution to this exists in the Cortix library, which makes it relatively easy to model a large, distributed system consisting of several discrete processes linked together. Our work involved the creation of a set of modules which would simulate the operation of a boiling water reactor, turbines and condensers, and then link them together using Cortix’s networking capabilities. This would serve as both a test of its modelling capabilities, as well as a basis for future work in balance of plant modelling as the modules can be used and implemented by other researchers within their own Cortix simulations.

Cortix Overview

Before going into depth on the design and interplay of the modules we created, we must describe in detail how Cortix works. The system itself allows several models to be implemented, connected and run through a series of time steps while passing information between each other. An example graphic is shown on the following page. Individual modules (the smaller squares), which are generally in the form of time-dependent differential equations with input and output values, are instantiated in the Cortix environment. These modules may then be connected together in a manner consistent with the system that needs to be modeled (e.g. a module simulating a nuclear reactor would connect to a turbine module and an inlet coolant pump module). The connections themselves are how modules transmit data to each other, with the information being passed generally being the output of a state equation of one module that is required for calculations by another module. The entire system runs within the Cortix environment, which handles the evolution of the modules themselves from their start to end times, as well as data passing between them through their connections.

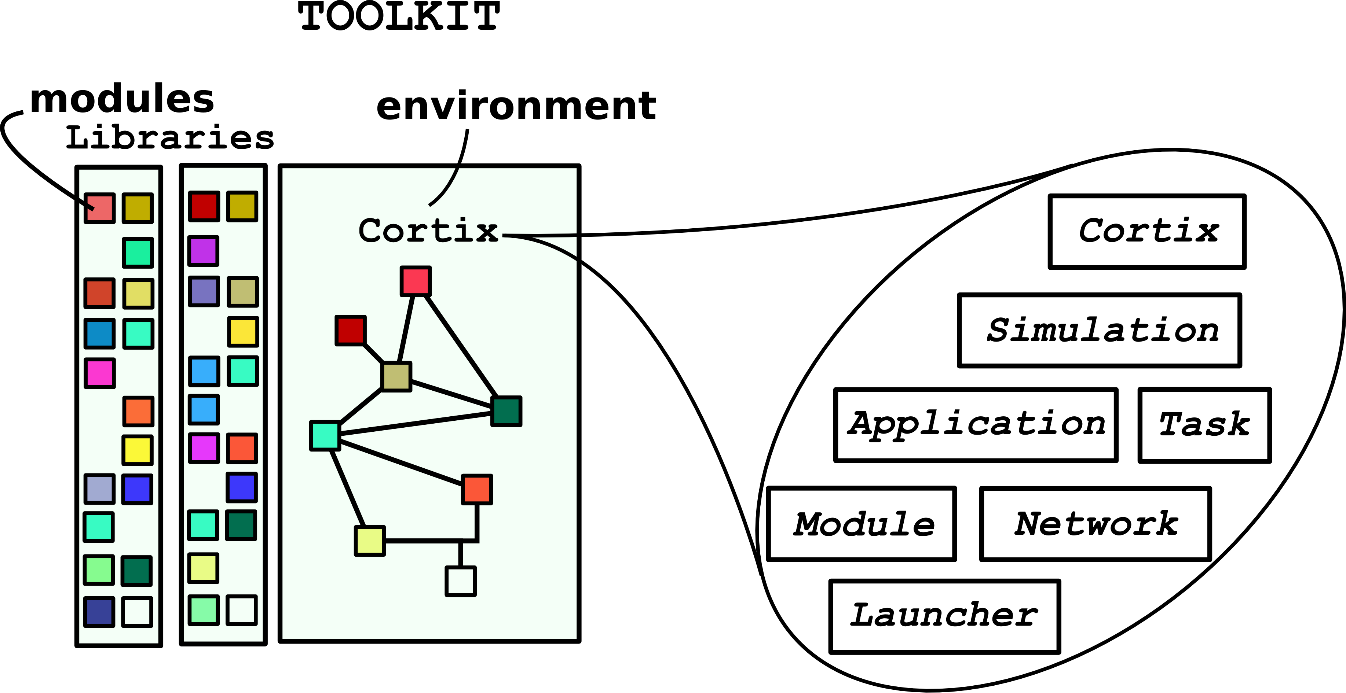


Fig. 1: The Cortix Environment[[4]](#footnote-4)

Each module within the Cortix environment is an instantiation of a class of modules representing a certain time of process or object (for example, a turbine class, a reactor class, a pump class, etc). These classes themselves are all subclasses of the abstract “Module” class, which provides the “send” and “recv” functions which are required to send data between the modules. Each subclass will also contain a set of parameters, internal functions and state variables which are unique to every class. The functions are typically used to calculate output or state variables based on some input data, which can then be sent to other objects for use in their calculations or stored in the module as a state variable in an object called a “quantity”. Quantities are Cortix-specific objects that use Pandas dataframes to easily store and retrieve data; each quantity contains data about the specific state variable’s name, unit, the name with which the unit should be graphed, and the value of the unit at every time that the Quantity is updated. Multiple different state variables can be tracked in the same Quantity, with each “row” of the quantity containing the information on each tracked state variable at a certain point in time, and each column being all of the values of a tracked state variable at all times. Singular points of data for a specific state variable can be retrieved at a certain point of time, or data for all tracked state variables can be retrieved by getting a single row of the Quantity. The Cortix Module superclass also contains methods to easily retrieve all values in the Quantity at all times and easily print and display them.

The Cortix environment itself is managed by the “run” file, which handles all aspects of simulation set-up, port connection, module evolution and data graphing. It will first import all the modules and libraries required for the simulation to run, and then set up storage objects which contain the parameters required by every module that will be instantiated. Then, a number of modules are instantiated as unique members of a specific Cortix module class, and are then connected to each other’s ports. The ports themselves are just lists of names kept by each module, which the run file then refers to in order to connect two ports together so that modules may pass data to each other. It does not matter if all the ports a module has are connected or not, the simulation will still run normally as long as a check is made to see if a port is connected before passing a send or receive function. At any rate, once all modules are instantiated and their ports are connected, the simulation will then run each of the modules on their own individual thread. The modules will evolve from their start time to their end time independently, as defined by their relevant internal functions (see below). They will pass data to each other independently, and when all the different modules have reached their end times then the simulation will end. At this point, graphs may automatically be generated by the run file from the quantities data stored by each module, and saved to the working Cortix directory.

The set-up described above is extremely powerful, as the end user only needs Cortix modules and to know which ports to connect to each other to run a simulation and obtain data. The Cortix back-end infrastructure handles the messy business of module evolution, multithreading, communication and data printing without any required user input besides declaring which modules will be used, their starting parameters and how they will be connected to each other. It is therefore possible for an end user to run a simulation using “black box” modules, the inner workings of which they are unaware of besides the names of the relevant quantities they want to view at the end of the simulation, the port names of the modules themselves and the names of the parameters that are required for successful simulation. This is also an extremely powerful feature of the Cortix environment, as with proper documentation the modules may be easily implemented by users running different simulations.

But how do the modules themselves work as they are evolved over a certain period of time? The answer to this is through each module’s run() function, which any properly implemented Cortix module must have defined by the programmer of the module. The internal implementation of the run() function can realistically be done in any manner, as long as there is some method by which the module will loop over a certain defined time span until the end-time defined in its parameters is reached. In the recommended implementation (which is used in the three modules that our group developed over the course of the semester), the run() function will contains a while loop which begins at the parameter-specified start time for each module; Then, it will run a “call\_ports” function, which sends and receives information to and from all its connected ports using the Cortix Module .send and .recv functions. It will then run the module’s own internal mathematical functions which are used to update the module’s state vectors, after which it will update the module’s current “time” by moving it forward one “time\_step”. Once the while loop has terminated (the module has reached its predefined end-time), the run() function will close, and once all open run() functions have closed simulation will halt and the modules Quantities may be accessed for graphing purposes.

We shall now discuss the specific implementation of the three modules that were created and implemented into Cortix modules by our group this semester, namely: bwr, turbine and condenser. Each of these contains mathematical models which calculate the state parameters (at this point, temperature, pressure, and flow quality) of the cooling fluid passing through them, as well as a few other variables which are dependent on the specific module.

The BWR Module

The first (and perhaps most important) module that we implemented is the one which models the neutron and heat transfer kinetics within a boiling water reactor via a point reactor model, called bwr. The variables it tracks as quantities are the prompt and delayed neutron densities within the reactor (saved in the neutron\_phase), and the average temperature of the nuclear fuel(saved in the reactor\_phase), with the temperature of the coolant leaving the reactor being an Additionally, functionality is given to send and receive data to other modules on the current coolant temperature within the reactor through its three (currently used) ports: coolant-inflow, coolant-outflow, and signal-out. coolant-inflow is where the reactor will get the state of the coolant entering the reactor (typically pumped in from a condenser), coolant-outflow is where the reactor sends the state of the steam that leaves the top of it (typically sent to a turbine), and signal-out is used to send data on the current state of the reactor to other unit operations within the balance of plant so that the coolant being sent back into the reactor arrives at the correct temperature and pressure for smooth operation.

The module itself uses a two-temperature point-reactor simulation method with moderator temperature feedback, part of which was modeled and programmed by Dr. V. F. Almeida. The internal helper functions which are used to model the kinetics of this reactor take as argument the temperature of the coolant entering the bottom of the reactor and use its own state variables (quantities) which keeps track of neutron density and fuel temperature to calculate the neutron density at the end of each time step and the temperature of the fuel and the coolant at the end of each time step as well. The internal logic proceeds as follows: First, within the run() loop, the internal function \_\_step() is called. The first thing this function will do is call all of the bwr module’s ports, either sending data to them using the send() function or receiving input data from the recv() function. Input data that is received is typically passed directly to the solver (more on that later), and output data is retrieved from the quantities history at time stamps specified by the module issuing the call for data from the bwr. Next, the module will build a vector of initial quantities that will be used to solve the differential equation that spans the current and next time step; this vector consists of the starting neutron and delayed emitter concentrations, the average temperature of the nuclear fuel and the average temperature of the coolant. Neutron and delayed emitter concentration densities, as well as the average fuel temperature are retrieved from the quantities data which is stored by the module itself, with average coolant inlet temperature being provided by the coolant-inflow port. Once this state vector is built, it will be passed to Numpy’s ode solver and solved numerically by the governing differential equations which apply to every value in the initial state vector.

The rate of change of neutron density with respect to time is given by the following equation:

Where is the reactivity within the reactor at any point in time t, is the delayed neutron fraction of 0.0065, l is the delayed neutron lifetime of about 0.1 seconds, is the sum of the decay rates of the six delayed neutron precursor groups, and is the strength of the neutron-emitting source inserted into the reactor during startup at any point in time t. is calculated by the internal function rho\_func, the inner workings of which are explained in further detail below. As represents the neutron source which is only inserted for a small amount of time during startup, it is modeled by an internal function q\_source() which simply returns roughly 10% of the maximum neutron density for times smaller than 1e-5 seconds, and 0 at all other times.

Each of the delayed neutron emitters is subject to the following balance, which is the differential equation that the solver uses to model their concentrations with respect to time:

Where i runs from 1 to six for each of the six delayed neutron precursors, is the total fraction of neutrons which that precursor makes up, is that precursor’s decay constant and is that precursor’s concentration at the current time, and n is the current neutron density at time t.

The rate of change of the average fuel temperature with respect to time is found by the following equation:

Where and are the density and heat capacities of the fuel, is the heat produced by the fuel due to nuclear fission per unit volume, is the rate of absorption of heat by the coolant and is the volume of the fuel. itself is calculated using the current neutron density at any point in time by the internal function nuclear\_pwr\_dens\_function, by the following equation:

Where is the heat energy released per fission event, is the number density of uranium-235 in the fuel, is the time-dependent neutron flux (simply the neutron density multiplied by average thermal neutron velocity), and is the fission cross section at the current fuel temperature. The fission cross section at the current temperature is calculated by sigma\_fis\_func, which uses the following equation:

Where is the original fission cross section at 293.15 K, is the reference temperature of 293.15 K and is the current temperature of the fuel which is passed to sigma\_fis\_func.

, the heat absorbed by the coolant, is calculated by a simple heat transfer overall heat transfer equation between the fuel and the coolant that is handled by the function heat\_sink\_rate:

Where is the overall heat transfer coefficient (specified by the parameters passed to the bwr module constructor), is the current temperature of the fuel and is the current average temperature of the coolant.

Finally, the differential equation governing the temperature of the coolant as it moves from the bottom of the reactor to the top of the reactor is found as follows:

With being the residence time within the reactor of the coolant, being the current temperature of the coolant (i.e. its temperature leaving the reactor), is the temperature of the coolant entering the bottom of the reactor, and are the density and heat capacity of the coolant and the volume of the coolant that flows through the reactor per residence time. Note that the calculation of all these equations is handled by a function called f\_vec which is passed to the ODE integrator, which takes as argument a vector containing the initial values of the 9 variables (neutron density, the six delayed precursor densities and the temperature of the fuel and the coolant) and calculates the output values after a full residence time for the numerical integrator, which runs these calculations until conversion is reached. The internal helper functions which are described above are called within the f\_vec function as needed.

The final major equations that pertain to the boiling water reactor module are the calculations which much be done to find the moderator temperature reactivity coefficient and the overall reactivity at any point in time. Overall reactivity itself is the sum of the effect of the position of the control rods and the coefficient due to moderator temperature reactivity, that is:

Where is the initial positive reactivity inserted into the reactor by withdrawing the control blades in order to start up, is the negative reactivity from the insertion of the control rods in order to control the reactor, is the moderator temperature reactivity coefficient, is the temperature of the moderator (or coolant, in the case of a bwr), and is the reference temperature for the moderator temperature reactivity coefficient (293.15 K). The reactor module itself displays some rudimentary control logic where is calculated and set in order to bring the reactor to the desired steady state neutron density by a more-or-less linear decrease in the overall reactivity (not including the effect of moderator temperature). All of these effects are calculated by the rho\_func() helper function, which is called in f\_vec() as a helper function. The derivation of itself is easily the most complex part of the set-up of the bwr module. It is given by a linearization of the rate of change of moderator temperature reactivity with respect to time, and is derived from the following equation:

Where  is the moderator temperature feedback coefficient, K is the neutron multiplication factor for a finite medium, and  is the temperature of the moderator. K is calculated as follows:

Plugging this in to the original moderator temperature reactivity coefficient equation leaves the following differential equation:

With being the fermi age or slowing down area, being the neutron diffusion length, being buckling, being neutron yield, being fast fission factor, being resonance escape probability and being thermal utilization. Only , and vary significantly with moderator temperature, with and being assumed constants over the range of temperatures. is found as follows:

With being the diffusion coefficient and equal to:

With being the macroscopic scattering cross section, and being a constant dependent on the material used as the moderator. is the neutron removal cross section, or the probability per unit length that a neutron becomes thermalized. It is calculated as:

With being the average logarithmic energy decrement for a collision involving a neutron and the moderator (in the case of a light water reactor, = 1), being the energy of a neutron produced during fission (assumed to be constant at around 2 MeV), and being the energy of the medium that the neutron is diffusing through, which is dependent on the temperature of the moderator by the following equation:

Where is the temperature of the moderator. Assuming that only Nm and vary with the temperature of the moderator, then the differential of with respect to may be calculated as follows:

Which expands to:

Next, the diffusion area in the moderator before absorption, , may be evaluated. It can be calculated through the following relation:

Note that L is referred to as the diffusion length whereas L2 is the diffusion area. Taking the derivative of the diffusion length, L, with respect to time, and noting that only the number density of the moderator varies with time, and that the macroscopic cross sections can be rewritten as Nm, we obtain the differential of diffusion length with respect to time:

Next, the terms in the four factor formula, must be developed and the derivatives of the ones which vary with moderator temperature must be calculated. and do not vary with moderator temperature, and are assumed to have constant values of 2.02 and 1.03. , the resonance escape, can be calculated from the following equation:

In the above equation, is the number density of the fuel, is the volume of the fuel, is the resonance escape integral, and Vm is the volume occupied by the moderator. With only the macroscopic scattering cross section of the moderator varying with temperature of the moderator (volume of coolant in the core is assumed constant, only the density changes), the derivative of the resonance escape probability with respect to the temperature of the moderator can be calculated as:

The thermal utilization, *F*, is calculated as follows:

With being the macroscopic absorption cross section of the fuel and being the macroscopic absorption cross section of the moderator. As only the macroscopic absorption cross section of the moderator will vary with the temperature of the moderator, the derivative of thermal utilization with respect to the temperature of the moderator can be written as follows:

The final preliminary differential equation is the one that must be developed for the differential of the moderator number density with respect to the temperature of the moderator. The change in number density of the moderator with respect to temperature will be proportional to the change in the mass density of the moderator with respect to temperature, by the molar mass of the moderator (in this case, 18 g/mol) and avogadro's number. The derivative of the mass density with respect to the temperature of the moderator of water is a function of a forty term polynomial used by the IAPWS 97 standards to approximate the behavior of water at different temperatures and pressures, that is:

The derivative can be approximated by using scipy's inbuilt differentiation function. From there, the differential of mass density with respect to moderator temperature may be related to the differential of number density with respect to moderator temperature by the molar mass and avogadro's number:

Now that all previous relationships have been developed, the moderator temperature feedback coefficient can be calculated. Plugging in all derived equations into equation 11 and taking the derivative with respect to the temperature of the moderator yields this final equation which can be used to approximate :

This final equation is used by the internal helper function alpha\_tn\_func() to calculate the moderator temperature reactivity coefficient at any point in time and with any moderator temperature. This function is called by rho\_func().

Once the ode integrator has successfully run over the specified time span, it will return ending values for neutron and delayed emitter densities and the ending temperature of the fuel and coolant. This information will be saved in the relevant quantities history at the end of the current time step for use in graphing and to send to other modules. Finally, the current time of the reactor (self.time) is advanced by one time step. After this is done, the \_\_step() function will terminate and the program will return to the top of the while loop in run() to either continue to the ending time or terminate the module.

The Turbine

The turbine class’ internal layout is much the same as the bwr class, with the exception that the parameters, port names, quantities and most of the specific internal helper functions will be different. The turbine class itself encapsulates a single turbine that takes in steam at a certain temperature, pressure, and quality and expands it to a final specified pressure and varying temperature and quality depending on the input parameters while also generating usable energy. Its two ports, inflow and outflow, representing steam coming in (which may be either saturated, superheated or a two-phase mixture) and the runoff from the turbine (which may be a subcooled liquid, a two-phase mixture, saturated or superheated steam, depending upon the properties of the mixture entering the turbine. The internal control logic proceeds almost exactly as bwr’s does, with the exception that within the step() function there is no ODE integration; instead, the internal helper function turbine() is called, which takes as argument the properties of the inflow steam received from the .recv() function in call\_ports(), and returns values to its quantities (turbine power, runoff quality, runoff temperature and the power generated by the turbine at that point in time).

The turbine() function itself is based around an energy balance on the steam entering and leaving the turbine. Starting from the relevant conservation equation, we have:

Where is removed from the system, is the shaft work that the turbine produces, and are the change in enthalpy, kinetic energy and height of the working fluid. Assuming an adiabatic turbine, and that the changes in height and kinetic energy of the fluid within the turbine are negligible, this leaves us with the following energy balance:

Noting that the change in enthalpy is equal to the difference between the enthalpy of the fluid at the inlet and outlet:

The inlet enthalpy is specified by the temperature, pressure and quality of whatever fluid is entering the turbine. The enthalpy of the fluid leaving the turbine can also be determined by an entropy balance and with the IAPWS python library. Assuming isentropic behavior, the entropy of the fluid leaving the turbine will be the same as the entropy of the fluid entering the turbine, that is:

With the entropy of the steam at the inlet being specified by its temperature, pressure and quality, this specifies the entropy of the steam at the outlet. Specifying the entropy at the outlet, and the pressure of the runoff leaving the turbine means that we can use the IAPWS functions to determine the temperature and enthalpy of the steam leaving the reactor. With the ending enthalpy of the steam known, the maximum possible isentropic work can be calculated by the following equation:

With being the mass flowrate of steam through the turbine. If the isentropic efficiency of the turbine is specified (some number between 0 and 1), then this can be multiplied by the total isentropic power to yield the actual change in enthalpy between the inlet and outlet streams:

With being efficiency, being the actual usable work/power the turbine produces with time, and being the actual ending enthalpy of the fluid leaving the turbine. With the ending enthalpy and pressure leaving the turbine known, the runoff temperature and quality can also be specified. The steam table can be used to determine if the runoff is a two-phase mixture; if it is, runoff quality can be calculated as:

Where is the enthalpy of the bubble-point mixture and is the enthalpy of the dew-point mixture at the outlet pressure. Once the runoff quality, pressure, temperature, and the power generated by the turbine is calculated, then the function returns these values and the step() function appends them to the relevant quantities history before incrementing the current time by one time step and repeating the while loop in run().

The Condenser

The condenser class has two ports: coolant-inflow and coolant-outflow. The inflow port will provide data on the two-phase mixture being passed to the condenser from the turbine (namely, quality and pressure), and the coolant-outflow port is used to send information on the temperature of the condenser runoff to the reactor. Standard Cortix module class rules apply, and the layout of the main functions run() and call\_ports() is more or less the same as in the bwr and turbine classes.

The function within the condenser which handles most of the condensing is the aptly named condenser() function, which takes as argument parameters relating to the state of the two-phase mixture being passed to it, and calculates the runoff temperature from the system. First, it calculates the amount of heat that must be removed from the turbine runoff to bring it to the state of dew-point liquid:

This will be used to determine how much the liquid is subcooled by. Then, the log-mean temperature difference between the turbine runoff and the cooling water at 287 K is calculated from the following equation:[[5]](#footnote-5)

Where and are the temperatures of the cooling water leaving and entering the condenser and is the critical temperature of the two-phase mixture entering the condenser. The temperature of the cooling water leaving the condenser at this stage may be determined based on the heat Q that is released by the two phase mixture as it condenses fully:

The total heat transfer coefficient between the two-phase mixture and the condenser can then be evaluated using the McNaught expression[[6]](#footnote-6):

is the overall condensation heat transfer coefficient, is the heat transfer coefficient of the pure liquid flowing through the condenser and is the Martinelli parameter, which may be calculated through the following relation[[7]](#footnote-7):

Where x is the quality of the mixture entering the condenser, and are the densities of the pure liquid and steam phases and and are the viscocities of the pure steam and liquid phases, respectively. is calculated for our uses through the Churchill-bernstein correlation for a pure liquid solution at a temperature equal to its critical temperature[[8]](#footnote-8). The Nusselt, Reynolds and Prandlt’s numbers can be very easily calculated using the IAPWS library. Once the overall heat transfer coefficient and log-mean temperature difference between the two-phase mixture and the cooling water is known, the amount of surface area required to condense the two-phase mixture into a dew-point liquid can be calculated as:

Once this is known, the area remaining for subcooling may simply be found by subtracting this number from the total heat transfer area in the condenser. Subcooling proceeds in a similar manner in the condenser: first, an ending temperature for the cooling water is guessed, and from this a heat balance is performed to determine the ending temperature of the subcooled liquid based on this initial guess:

Where is the cooling water mass flowrate, is the heat capacity of the cooling water, is the difference between the exit temperature of the cooling water leaving the condensation stage and the temperature of the cooling water leaving the subcooling stage, is the mass flowrate of the process fluid (the two-phase mixture being condensed), is the heat capacity of the process fluid and is the dew-point temperature of the process fluid. From this, the log-mean temperature difference may be calculated as:

Where coolant refers to the cooling water and process refers to the process fluid (I.E. the actual condensate). With guesses for the starting and ending temperatures of the process and cooling fluids, the average heat transfer coefficient can then be determined by the Churchill-Bernstein correlation. A guess for overall heat transferred between the two fluids may then be calculated by the following equation:

With being the calculated overall heat transfer coefficient, A being the area remaining for subcooling (calculated with the result from equation 38), and being the total heat transferred between the two fluids by our “guess”. From this guessed heat transfer, we can then back-calculate the final temperature of the process fluid and the cooling water:

And from this, another value for may be calculated from equation 39 by plugging in the new value for . These new values may then be plugged back into equations 40, 41 and the Churchill-Bernstein correlation to generate a new , with this process being repeated until convergence is reached and final approximate values for and are generated. is then returned by the condensing function, and is appended to the condenser runoff quantities history so that it can be given to the bwr module as an input variable in the next time step.

Results

Our group constructed a simulation consisting of a condenser, a bwr and two turbines laid out in the following configuration (which is conveniently automatically generated by the Cortix software):

A close up of a logo

Description automatically generated

Fig. 2: The Networking Used By Our Simulation

As can be seen by the graphic, a boiling water reactor module boils up steam and passes this to a high pressure turbine’s inflow port. The turbine expands the steam from whatever pressure enters the turbine to about 0.5 mPa, and then passes this off to two low pressure turbines which further expand the steam to about 0.5 bar. This is then passed on to the condenser, which subcools the low pressure turbine runoff and passes it back into the bwr to be boiled up again. Our group ran a simulated startup to examine its behavior.

During startup, all three of the turbines are “tripped” until a certain reactor pressure is reached to allow pressure to build up in the reactor. The reactor itself initially receives a positive reactivity insertion as well as a kick from an initial neutron source. The source is quickly removed, and then control rods gradually drop into the reactor to reduce reactivity until the desired steady-state neutron concentration is reached. Once this occurs, thermal equilibrium is reached between the fuel and the coolant, and the system settles in to its steady state values.

A close up of text on a white background

Description automatically generated

Fig. 3: Reactor Neutron Density

The neutron density within the reactor follows the expected s-shaped curve, with an initial delay before a quick rise in neutron density to the steady state value of 1 (which is a dimensionless number representing the ratio of current neutron density to desired steady-state neutron density which makes it easier for the numerical solver to work with). As neutron density approaches the steady state value (1), it levels off as the control rods drop in and bring total reactivity to zero.

A close up of a map

Description automatically generated

Fig. 4: Delayed Neutron Emitter Concentrations

The concentrations of the delayed neutron emitters also follows the expected s-shaped patterns, although they differ in height due to their different decay constants which limit them to certain steady-state concentrations. The numbers on the y-axis here are also relative, this time to the smallest delayed neutron emitter concentration (so the highest emitter concentration is about 450x more abundant than the lowest delayed emitter concentration). The shape of the curves is exactly as we would expect; however, the actual ratios seem to be a bit off, and further investigation will need to be made to ensure that the delayed neutron emitter balance is set up properly.

A close up of text on a white background

Description automatically generated

Fig. 5: Average Fuel Temperature

The temperature of the fuel displays an initial slight jump in temperature from its initial value of around 293.15 K to a little over 300 K as the neutron source is inserted, causing a small amount of fission and an increase in temperature. As the neutron source is removed, the temperature stops increasing until neutron density begins to pick up at around five minutes, after which time it rapidly increases to its ending average value of around 930 K. The small discontinuity just before the fuel temperature reaches steady state is due to an error in the condenser that causes its output temperature to drop by about 10 K that we were not able to fix in time for the final presentation. This sudden decrease in inflow temperature initially brings the temperature rise in the reactor to a halt before continuing to increase later on to its final steady state value.

A close up of text on a white background

Description automatically generated

Fig. 6: Temperature of The Steam Drawn Off The Top of The Reactor

The temperature rise of the coolant follows more or less the same pattern as that of the fuel, which is as we would expect since their heat balances are so closely connected. A final temperature of around 430 K is a bit low but still in the realm of possibility. Another interesting thing to note is that the drop in the coolant temperature after the drop in the condenser runoff temperature is much sharper then the drop in fuel temperature at this same period of time. This is due in part to the fact that nuclear heat production has almost reached its steady-state value by the time this occurs, and so a slight decrease in the coolant temperature will therefore result in an increase in the heat transferred to the coolant from the fuel and a slowdown of further heat increases. There will however be an immediate 15 degree drop from the coolant due to the decrease in the temperature of the condenser runoff.

A close up of a piece of paper

Description automatically generated

Fig. 7: Condenser Outflow Temperature With Respect To Time

The condenser reaches its initial steady state value of a runoff temperature of about 307 K and remains there for a short period of time. Due to a bug in code execution, it then drops about 20 degrees to a final steady state value of about 287.5 K. The drops are quick, but not immediate, proving that the iterations used to calculate turbine runoff have at least some efficacy.

A close up of text on a white background

Description automatically generated

Fig. 8: High Pressure Turbine Power With Respect To Time

The high pressure turbine and low pressure turbines have the same graphs for power due to another bug in execution that couldn’t be ironed out before the due date, and so only one is presented here. Power generated by the turbine rises extremely quickly due to the sensitivity to the temperature of the steam entering the turbine (as it is generally a two-phase mixture, increases in temperature will also result in an increase in pressure and a correspondingly high enthalpy increase). This can also be seen in the large falloff in turbine power around 12 minutes due to a decrease of only around 20 degrees K in the temperature of the steam being drawn off the top of the reactor (see Fig. 6). Final steady state power generation of about 80 megawatts is also within the realm of possibility for a reactor operating at temperatures of about 450 K and with steam flowrates of 1820 kg/s.

A close up of text on a white background

Description automatically generated

Fig. 9: High Pressure Turbine Outflow Temperature

Outflow temperature gradually increase from the high-pressure turbine as the inlet temperature increases. Outflow state will begin as a subcooled liquid, and then increase in temperature until it becomes a two-phase mixture at the critical-point temperature at 0.5 mPa, at which point only outflow quality will continue to increase. The initial plateau in outflow temperature is seen to be consistent with the initial plateau in reactor runoff temperature as the effects of the initial neutron insertion become lost.

A screenshot of a cell phone

Description automatically generated

Fig. 10: Low Pressure Turbine Outflow Temperatures

Note that the above figure’s results are consistent with the results for the second low-pressure turbine as well, which has been omitted for simplicity’s sake. The two-phase mixture being passed from the high pressure turbine to the low pressure turbines has enough enthalpy to immediately bring the outflow from the low-pressure turbine to two-phase mixture; It remains at the critical temperature around 50 kPa for almost the entire simulation, with the only major runoff changes that would be observed being a gradual increase in the quality of the runoff as the inflow comes in at a higher and higher enthalpy.

Summary and Conclusions

Overall, besides a bug in how the condenser and low-pressure turbines functioned, results obtained from this network simulation were at least theoretically possible, although at this point in time we do not have hard validation with other sources/simulations. What we do have, however, is a set of modules implemented in a network which can serve as the basis for future forays into modelling the nuclear reactor’s balance of plant. Our work proves that effective nuclear network simulation can be performed with Cortix and further expanded into more complex networks and modules if given more time.

Acknowledgements

We would like to sincerely thank Dr. V. F. Almeida for the abundance of help he has given us. Without his help with debugging our code and his knowledge of the Cortix networking itself we would not have been able to implement and create the network that we currently have. His two-temperature point reactor model also served as the basis for the model which we have implemented into Cortix.

Without jjgomera’s IAPWS Python library, it would have been impossible to develop functions accurately modelling the behavior of the cooling water within the reactor, turbines and pumps. The functionality that his library provides to calculate almost any thermophysical property of water at nearly any temperature-pressure combination proved invaluable to our modelling efforts.

Bibliography

1. [↑](#footnote-ref-1)
2. [↑](#footnote-ref-2)
3. [↑](#footnote-ref-3)
4. [↑](#footnote-ref-4)
5. [↑](#footnote-ref-5)
6. [↑](#footnote-ref-6)
7. [↑](#footnote-ref-7)
8. [↑](#footnote-ref-8)