Development of a Pressurized Water Reactor System Operation and Accident Dynamics Analysis Program

Submitted by

Examination Roll: 142307

Registration Number: 2018025822

B.Sc. Admission Session: 2018-19

B.Sc. Academic Session: 2021-22

Department of Nuclear Engineering,

Faculty of Engineering and Technology

University of Dhaka

This dissertation is submitted in partial fulfillment of the requirements for the degree of **Bachelor of Science** in **Nuclear Engineering**

February 28, 2024

Declaration

I, Ebny Walid Ahammed, hereby declare that this thesis, entitled "Development of a Pressurized Water Reactor System Operation and Accident Dynamics Analysis Program," is entirely my own work under the supervision of Saad Islam, Lecturer, Department of Nuclear Engineering co-supervision of Md Ali Mahdi, Lecturer, Department of Nuclear Engineering, University of Dhaka. It has not been submitted in whole or in part for any other degree. All sources of information and material I have used, whether directly quoted or paraphrased, have been duly acknowledged through proper citation and reference. I also affirm that the research presented in this work is original and that all data and findings are authentic and accurate to the best of my knowledge.

Additionally, I confirm that any contributions made by others to this work, including but not limited to research assistance, editing, and proofreading, have also been duly acknowledged in the appropriate sections of this thesis. I understand that any act of academic dishonesty, including plagiarism or falsification of data, is unacceptable.

Ebny Walid Ahammed
Examination Roll : 142307
Registration Number: 2018-025-822
Session: 2018-19
Department of Nuclear Engineering
Faculty of Engineering and Technology
University of Dhaka

Certificate of Approval

The thesis titled **Development of a Pressurized Water Reactor System Operation and Accident Dynamics Analysis Program** is done under my supervision, meets acceptable presentation standards and can be submitted for evaluation to the Department of Nuclear Engineering, University of Dhaka in partial fulfilment of the requirements for the degree of *Bachelor of Science* in **Nuclear Engineering**.

Supervisor

.....

Saad Islam

Lecturer
Department of Nuclear Engineering
University of Dhaka
Dhaka-1000, Bangladesh

Co-Supervisor

.....

Md. Ali Mahdi

Lecturer
Department of Nuclear Engineering
University of Dhaka
Dhaka-1000, Bangladesh

Acknowledgements

Embarking on this journey would not have been possible without the exceptional guidance and support of several exceptional people. My deepest gratitude goes first and foremost to Saad Islam, my mentor throughout this project. His expertise and insight proved invaluable, shaping my graduate experience every step of the way. From the initial spark of the project to the final strokes of my thesis, his unwavering support and encouragement kept me motivated and pushing forward.

A special note of thanks goes to my co-supervisor, Md Ali Mahdi. His contagious enthusiasm for the project inspired my own, and his dedication to bridging the gap between theoretical concepts and real-world applications broadened my understanding of what engineering truly means. His programming guidance was just the tip of the iceberg; he taught me invaluable lessons that extend far beyond technical skills.

My journey wouldn't have been complete without the unwavering love and support of my family. They showered me with encouragement and belief. Their presence fueled my determination and helped me turn a long-held dream into reality.

Finally, I would also like to thank my friends Mahmudul Hasan Shuvo , Tasfia Rahman Riva and Sabyasachi Roy . Their support and encouragement were invaluable, especially during challenging times. I wish nothing but success in their lives. This project stands as a testament to the incredible power of kindness and support. To all who played a part, I offer my heartfelt thanks.

Dedication

To my beloved family
Ammu Asmotara, Baba Md Safir Uddin, my younger bother Tauhid Hasan, Dada bhai Alamin Khalid and Didi bhai Tamanna Tanha.

Abstract

This thesis presents the development of a comprehensive nonlinear dynamic program for a PWR system, offering valuable insights into its transient behavior and paving the way for robust control strategies. The core of the model lies in the intricate interplay between neutronics and thermal hydraulics within the reactor core. Point kinetics equations govern the reactor power dynamics, utilizing six lumped neutron groups for computational efficiency. Coupled with this are thermal-hydraulic considerations, accounting for heat transfer from fuel to coolant via an overall heat transfer coefficient and employing a single-phase natural circulation model to capture the inherent buoyancy-driven flow. Beyond the reactor core, the model delves into the nuances of other crucial components. The U-tube steam generator is meticulously represented by a single-tube depiction with dynamic boundaries encompassing sub-cooled, boiling, and superheated regions. This intricate approach captures the complexities of heat transfer during phase change and accurately reflects the steam generation process. Maintaining optimal pressure within the system is critical, and the pressurizer model steps in. This model adheres to the fundamental principles of conservation of fluid mass, volume, and energy, ensuring a realistic representation of pressure dynamics during transients. Additionally, the hot leg riser and down comer are incorporated as first-order lags, accounting for their influence on coolant flow and pressure distribution. To encompass the complete picture, the PWR model is coupled with a steam generator model, providing feedback on reactor power based on steam flow, pressure, and enthalpy inputs. This crucial connection allows for a comprehensive analysis of the entire energy conversion process. The true prowess of the model lies in its ability to faithfully replicate real-world scenarios. Simulations were conducted for various perturbations, including control rod withdrawal and changes in steam demand. The model's predictions were then meticulously compared against design data, showcasing remarkable agreement, particularly at full power. This serves as a testament to the model's accuracy and predictive capabilities. Furthermore, the model extends its reach beyond mere simulation. Steady-state control programs were implemented for both reactor power and pressurizer pressure, demonstrating their effectiveness in maintaining stable operational conditions. This paves the way for the development of advanced control systems capable of ensuring the safe and reliable operation of PWR reactor systems.

Contents

List of tables

List of figures

Chapter 1

Introduction

A nuclear power plant is a very complex system. Understanding the operational mechanism of a power plant is not only the sole job of the reactor operator but also all the engineers associated with the power plant as every single component of the power plant operates in a collective behaviour.

1.1 Motivation

Bangladesh is stepping into the nuclear club with the inauguration of the Rooppur Nuclear Power Plant to meet its energy demand per capita. Rooppur nuclear power plant will be a milestone to prove Bangladesh's commitment to carbon-neutral energy and it will also help us to meet the energy demand which is predicted to rise in the future [1].

Nuclear power will be a great solution to global warming and climate change issues and ensures that the operations of existing and future nuclear power plants will be safer so that any other Fukushima or Chernobyl in Bangladesh. The power plant authority and their guardian organization, the Bangladesh Atomic Energy Commission (BAEC) need to ensure the best of the human resources they can find.

The training of the nuclear reactor operators can be done in two ways.

- Experimentation and hands-on training.
- Simulations of the system

Due to the legal complexities of the International Atomic Energy Agency (IAEA) and the licensing requirements for reactor operators, practical experimentation and hands-on training are not feasible for gaining a better understanding of the system. Therefore, computer simulation emerges as the most effective method for this purpose. Most simulators for reactors or nuclear power systems feature a user interface (UI) that closely resembles the control room, preparing operators to manage the actual reactor in real time. Fig-

.

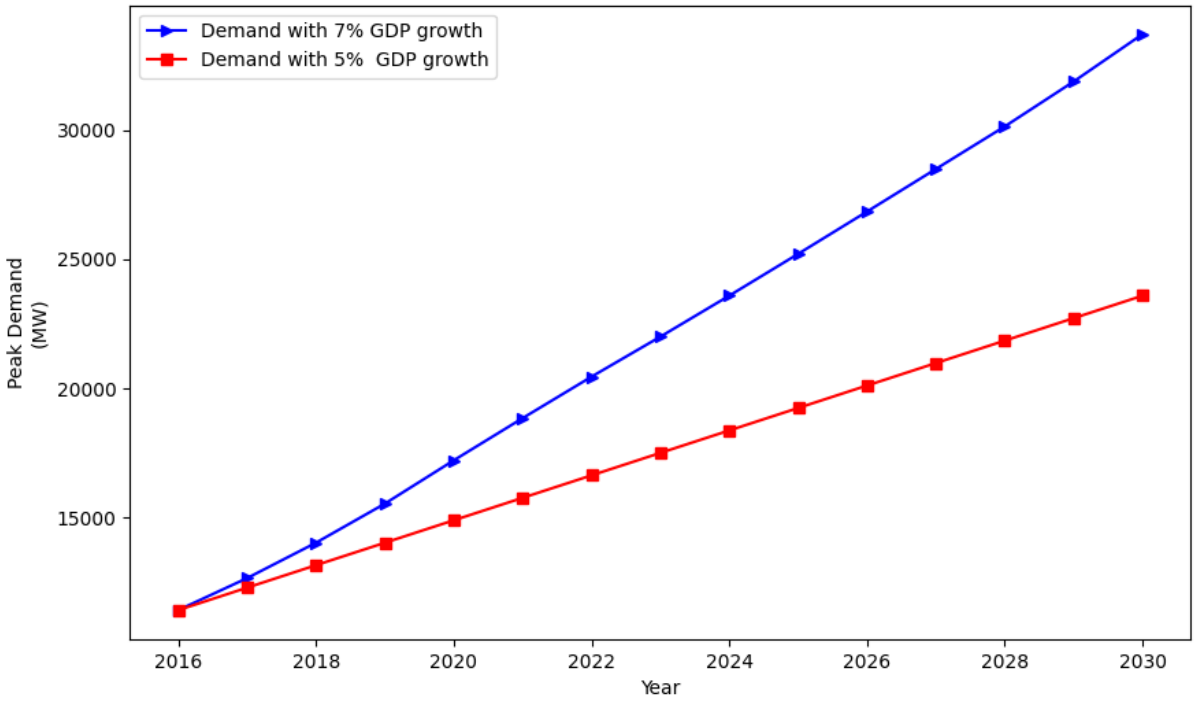

Figure 1.1: Energy Demand based on the GDP growth [1]

ure 1.2 shows the UI system of PCTRAN (Personal Computer Transient Analyzer), a PC-based Nuclear Power Plant Simulator developed by Micro-Simulation Technology.

Nuclear power plant operators play a crucial role, necessitating exceptional training and evaluation to guarantee safe and efficient operations. Training simulators are pivotal in this context, as they effectively replicate plant behaviour under both normal and extraordinary circumstances. This feature is essential for refining operator skills and setting proficiency benchmarks. The Chernobyl disaster highlighted the severe implications of human errors in nuclear operations, leading to a worldwide initiative to improve operator performance. Consequently, strict regulations were enforced, and the adoption of Nuclear Power Plant Training Simulation Systems (TSSs) became widespread. These comprehensive simulators act as virtual duplicates of actual plants, accurately mirroring the intricate interactions of systems, variables, and responses. Industry norms dictate that TSSs must exhibit high fidelity and predictive accuracy. They are required to precisely simulate reactor dynamics, control system behaviours, and emergency protocols. This enables operators to confront and manage a variety of scenarios in a risk-free and controlled setting. Such immersive training promotes in-depth understanding, enhances critical thinking, and hones quick decision-making skills. These attributes are crucial for the safe and reliable functioning of nuclear facilities [3].

Beyond replicating normal and abnormal operations, TSSs can also be deployed for various other purposes. They can be used to test new procedures and equipment, validate safety assessments, and conduct research on human-machine interaction in complex

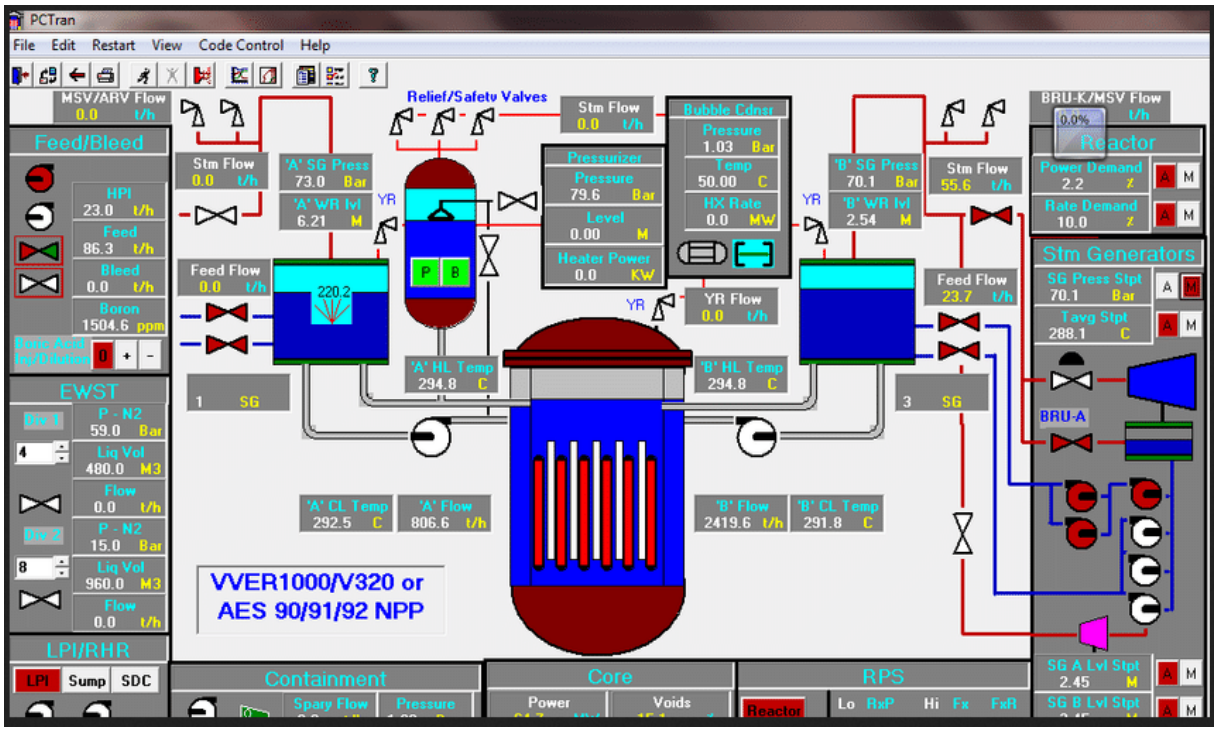

Figure 1.2: PCTRAN VVER1000 Micro-Simulation [2]

systems. This versatility further underscores the immense value of training simulators in ensuring the safe and efficient stewardship of nuclear power. In essence, TSSs represent a cornerstone of modern nuclear plant operation. By providing a platform for rigorous training, continuous assessment, and proactive preparation, they empower operators to handle both routine and unforeseen challenges with the highest degree of competence and confidence, ultimately safeguarding the well-being of communities and the environment [3].

The market value of these simulators is very high and as of now, there are no open-source simulator kernels or programs offering an Application Programming Interface (API) that facilitates the computation of power plant dynamics. This study aims to address these problems,

- Open source nuclear power plant simulator software kernel development
- an API-based nuclear power plant operation and accident dynamics program development
- An operation and accident dynamics program that can be an alternative to **PCTRAN** but free and more customized.
- An operation and accident dynamics program that can be applied to investigate the operation and accident dynamics of any PWR type power plant.

- An operation and accident dynamics program that can generate the necessary training data for an AI-powered alarm system in a nuclear power plant.
- An operation and accident dynamics program that implements multi-threading capabilities in the central processing unit (CPU).

Chapter 2

Literature Review of prior work

Training nuclear power plant operators is crucial and can be approached in two ways: either through hands-on experience in an operational plant or through computer simulations using advanced algorithms. Operating an actual nuclear power plant for training purposes is often restricted by legal and regulatory constraints. Therefore, the preferred method is usually conducting computer simulations of nuclear systems. These simulations are used to train personnel in a virtual environment. To create these virtual systems, professionals typically rely on dynamic modelling, which is not a new concept in the nuclear industry. However, most of the existing models are somewhat dependent on the specific design and construction features determined by the manufacturer.

2.1 Previous work on the dynamic modelling:

Masud et al [4] studied a comprehensive nonlinear simulation model of a typical Pressurized Water Reactor (PWR) system. The significance of this model lies in its ability to capture the effects of nonlinear processes, such as two-phase flow, on the dynamic behaviour of a PWR system—something that linear models fail to achieve. By introducing artificial anomalies into the plant components, the model facilitates the identification of the behaviour of various plant variables within a nonlinear framework. Additionally, a linear simulation was conducted to establish a linear operational range for the primary loop components. This was then juxtaposed against the nonlinear simulation for these same components, providing valuable insights into the limitations of linear simulation methods in accurately predicting system behaviour under varying conditions. The research also extends to the development of specific subsystem models, including those for the steam generator, pressurizer, and turbine. These models have broader applications, such as in simulating components of fossil fuel power plants and the creation of diagnostic neural networks. Moreover, the computer codes developed for this study are self-contained and crafted using advanced simulation software languages like ACSL and MATRIXx. This

design choice enhances the versatility of the models, allowing for easy modification with new plant data or further system development. The scope of this work has potential applications in conducting accident and safety analyses for nuclear plants, as well as identifying plant variables that are particularly sensitive to component anomalies. This comprehensive modelling approach provides a robust framework for studying and understanding the complex dynamics of PWR systems.

Samet Egemen Arda [5] in his Ph.D. thesis developed a nonlinear dynamic program for the analysis of a passively cooled SMR reactor. This study includes a mathematical model of a fast nuclear reactor core along with detailed thermal-hydraulic modelling. There are also models of a helical steam generator and a pressurizer. He also studied the control elements which include control rod movement, spray, and surge flow whenever any pressure oscillation is observed in the reactor core due to the temperature variation in different coolant nodes. This study also showed the dynamic implementation of these mathematical models using Simulink (a popular Matlab library). This study was a good demonstration of how design and modelling can be a good way to predict nuclear reactor dynamics.

Mohamed Rabie Ahmed Ali [6] conducted a comprehensive study and developed an intricate model of a recirculating type U-tube steam generator in his PhD thesis. He adopted a lumped parameter approach for an in-depth investigation of a U-tube steam generator. In his research, Ali developed four distinct models, labeled as models A, B, C, and D, corresponding to 3, 5, 7, and 14 nodes, respectively. He further validated these models against experimental data from the H. Robinson nuclear power plant. His findings indicated that model D exhibited superior performance in terms of achieving steady-state data efficiently over time. The mathematical modelling and findings from Ali's study were subsequently utilized in the work of Masud et al. [4] as the base model for the steam generator. Additionally, the assumption of linearity between the water properties in Ali's study has been widely adopted in subsequent research.

Alramady et al [7] also investigated the dynamic behaviour of a U-tube steam generator. Their paper presents a comprehensive review of a U-tube steam generator (UTSG) model, meticulously developed within the MATLAB environment. The UTSG model is intricately structured into four distinct regions: the upward and downward primary regions, and the upward and downward metal tube regions, accompanied by secondary regions. These secondary regions are further categorized into heat transfer, steam separation, and sub-cooled water regions. The foundation of the model lies in the application of energy and mass conservation equations across all these regions, ensuring a holistic and accurate representation of the system's dynamics. A notable advancement in this model is the introduction of precise functions that adeptly characterize the thermodynamic properties of saturated steam. This approach marks a significant departure from the conventional interpolation methods commonly employed in such models, promising

enhanced accuracy and reliability. The paper not only delves into the steady-state analysis of the UTSG model but also ventures into examining a specific transient case, providing a comprehensive understanding of the system's behaviour under varying conditions. This review underscores the model's potential in revolutionizing UTSG simulations, owing to its robust framework and innovative methodologies.

Vineet Vajpayee et al [8] also studied the dynamic modelling of nuclear power plants. They provided comprehensive modelling on the development of an integrated non-linear dynamic model for a Pressurized Water-type Nuclear Reactor (PWR) and its associated components. This model is primarily designed for control design and evaluation purposes, employing a first-principles approach for a realistic representation of the nuclear plant's components. The review delves into the intricacies of modelling the dynamics of critical reactor components, including the reactor core, thermal hydraulics, piping and plenum systems, pressurizer, steam generator, condenser, and turbine-governor system. It emphasizes the significance of incorporating various actuators and sensors to create a holistic model that accurately mimics the operational dynamics of a PWR. A key focus of the review is the testing and validation of the proposed model. This is achieved by introducing perturbations in different input variables, thereby assessing the model's response and resilience under varied operational scenarios. The review also highlights the implementation of various control loops within the model. These loops utilize low-level Proportional-Integral (PI) control strategies, integral for simulating the closed-loop behaviour of the nuclear plant. Specific control loops are dedicated to managing reactor power, steam generator pressure, pressurizer pressure and level, and turbine speed, ensuring comprehensive coverage of all critical aspects of PWR operation. Further, the review discusses the advancement of control strategies with the incorporation of Linear Quadratic Gaussian-based optimal control techniques. These strategies are noted for their enhanced efficacy and precision in managing complex nuclear reactor operations. Unique to this work is the extensive coverage of various plant sections, the meticulous implementation of finely tuned control strategies, the completeness and detail of the model equations, and the availability of parameter values. These aspects are critically analyzed, underscoring their contribution to making the model not only implementable but also a potential benchmark for future control design studies in PWR nuclear power plants. In summary, this literature review provides a thorough analysis of the innovative approaches in modelling and control design specific to Pressurized Water-type Nuclear Reactors. It encapsulates the strides made in this field, offering valuable insights into the potential applications and advancements in nuclear reactor control and safety systems.

Dutta et al [9] studied the detailed modelling and dynamics of a nuclear steam turbine. Their predictive model has been developed to forecast high-pressure and low-pressure turbine torque during thermodynamic transient conditions. The model is grounded in the principles of mass and energy conservation, encompassing critical components such

as the high-pressure turbine, low-pressure turbine, feed heaters, re-heater, and moisture separator. The simultaneous solution of dynamic equations yields stage pressures across different load conditions. The mathematical formulation of the model and open-loop responses to specific disturbances are provided in detail.

Hu et al [10] studied a dynamic mathematical model for the simplified steam condenser. In their study, they have included the details of modelling starting from the cold water side to the steam level in the hot water well. They employed the lumped parameter modelling approach. Subsequently, a pressure Proportional-Integral (PI) control system for the steam condenser is developed using the Matlab/Simulink simulation platform. To optimize controller performance, the grey wolf optimizer (GWO), a novel meta-heuristic intelligent algorithm, is employed for fine-tuning the PI controller parameters. Comparative analyses are conducted with other tuning methods, including the Ziegler-Nichols (Z-N) engineering method, genetic algorithm, and particle swarm algorithm. Simulation results demonstrate that the GWO algorithm exhibits superior control performance compared to the other four algorithms.

Klimanek et al [11] explored the numerical modelling of natural draft wet-cooling towers. The development of a comprehensive Computational Fluid Dynamics (CFD) model was outlined, showcasing its capability to predict cooling tower performance across diverse operating conditions. The methodology adopts a step-by-step process, commencing with the detailed presentation of one-dimensional numerical modelling for counter-flow heat and mass exchangers. Subsequently, the paper delves into reduced-order modelling, specifically employing proper orthogonal decomposition radial basis function networks, to capture heat and mass transfer within cooling tower fills. The discussion then extends to the modelling approaches for droplet zones, encompassing rain and spray zones. The paper concludes by addressing two and three-dimensional CFD models, offering a holistic perspective on the numerical modelling of natural draft wet-cooling towers.

Li et al [12] utilized experimental data from the floating nuclear power plant (FNPP) pump, this paper constructs a comprehensive characteristic curve for the pump through fitting and extrapolation methods. The paper also establishes a speed calculation model for the pump within the RELAP5 code. Using this simulation model, various response processes, including steady-state conditions, starting characteristics, coast-down characteristics, and direct switching conditions between high-speed and low-speed, are systematically analyzed. The findings indicate that the pump model effectively fulfils its simulation function, accurately reflecting the operational characteristics. Notably, the FNPP pump exhibits a substantial demand for inertia, necessitating appropriate design considerations based on actual conditions. The developed pump model proves to be applicable for subsequent analyses of diverse responses within the FNPP reactor system. The results obtained offer valuable insights for pump design and optimization, providing a reference point for future endeavours in this field.

Gao et al [13] studied the accurate calculation of rapid flow transients in a reactor coolant pump system held significant importance for the safety analysis of a nuclear reactor. Specifically, an in-depth transient analysis of flow coast down was crucial for the design and manufacturing processes of a reactor coolant pump, as the reliable operation of such a pump was essential for ensuring the safety of a nuclear power plant. The paper presented a mathematical model designed to solve for flow rate and pump speed transients during the flow coast-down period. Remarkably, the model did not necessitate detailed information on the centrifugal pump characteristics. In that study, the analytic solution for the non-dimensional flow rate revealed its dependence on the energy ratio $\check{v}$. This parameter ($\check{v}$) encapsulated the kinetic energy of the loop coolant fluid and the kinetic energy stored in the rotating parts, making it a crucial factor in determining the non-dimensional flow rate. Additionally, when the steady-state flow rate and pump speed remained constant, the inertia of the primary loop fluid and the pump moment of inertia emerged as important parameters in flow transient analysis. Under the condition where all pump shafts were seized, the flow decay was influenced by the inertia of the primary loop fluid. In scenarios where the pump inertia was substantial, the flow decay was primarily determined by the pump inertia. The calculated non-dimensional flow rate and non-dimensional pump speed using the model were compared with experimental data from two nuclear power plants and a reactor model test focused on flow coast-down transients. The comparison demonstrated good agreement, although as the flow rate approached zero, discrepancies between experimental and calculated values increased due to the influence of mechanical friction loss.

Chapter 3

Methodology

This chapter is dedicated to the overview of a standard pressurized water reactor power plant and outlines a method for creating precise models of its various components. Additionally, it outlines the fundamental design processes for the control systems utilized within a typical PWR system. The chapter explores PWR models and associated control systems comprehensively.

3.1 Pressurized Water Reactor Power Plants

Pressurized Water Reactor (PWR) power plants, whether third or fourth generation, commonly employ a cooling system with two distinct circuits. The first circuit, a closed-loop system, involves the primary coolant—usually water—flowing through the reactor core, serving as both a moderator and a coolant. This primary coolant maintains a constant pressure as it extracts heat from the reactor core. The secondary circuit, starting at the steam generator inlet, receives the heat from the primary coolant, causing a phase change to saturated vapour. This vapour then traverses through a turbine, converting thermal energy into mechanical energy, and subsequently enters a condenser where it condenses back into liquid form. The condensed liquid is then pumped back to the steam generator inlet, closing the loop. This dual-circuit configuration enables efficient heat transfer and electricity generation, with safety features incorporated to ensure the secure operation of the nuclear reactor. Fig 3.1 shows a diagram of a three-loop PWR power plant.

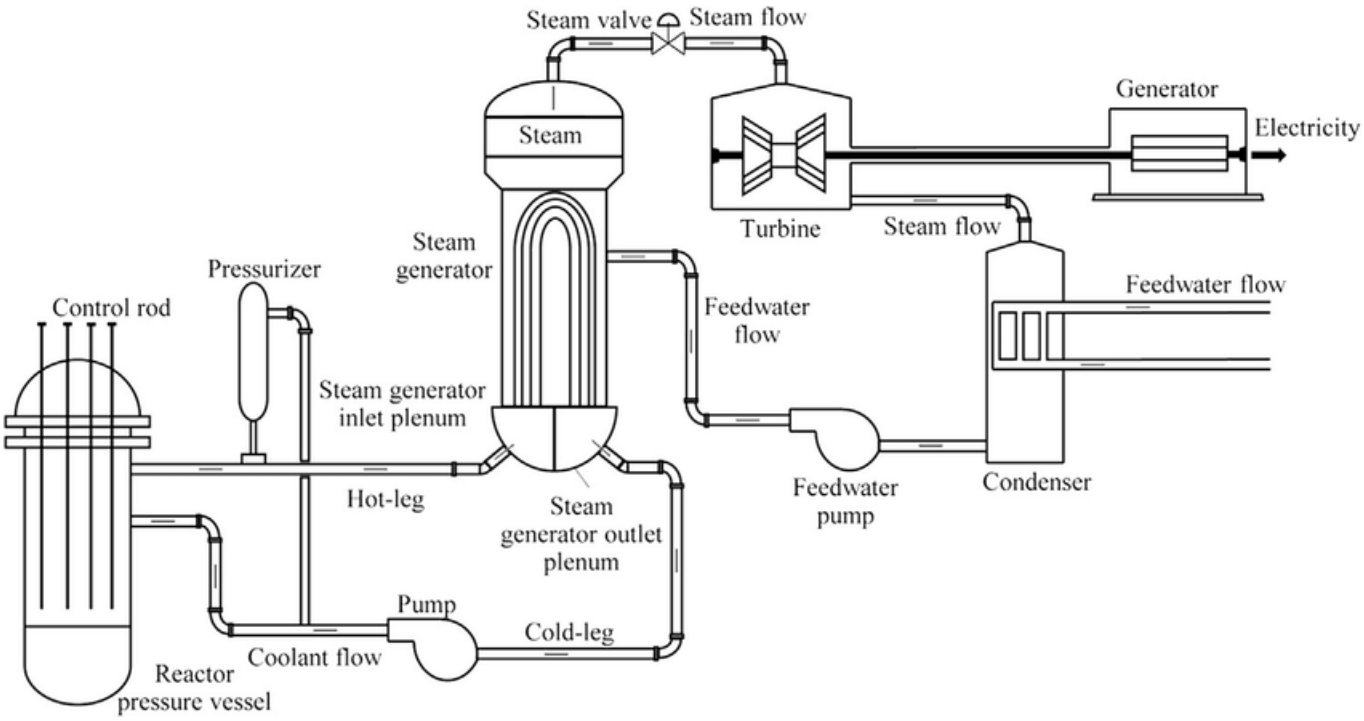

Figure 3.1: pressurized water reactor system [14]

3.2 Modeling Procedures

The adopted approach in this study involves employing a lumped-parameter modelling technique as the foundation for system modelling which was also done in Naghedolfeizi et all[4]. In this approach, the typical performance of the model is evaluated by considering average characteristics across a consolidated area or region. In this methodology, the average behaviour of the model is assessed by assuming average properties over a lumped region. As spatial dependencies are disregarded, the describing equations are formulated using ordinary differential equations. These equations are derived from physical laws, empirical observations, and constitutive relations relevant to the system. The process of developing a model for a physical system using a lumped-parameter approach encompasses several key steps. Initially, the component to be modelled is chosen, followed by the specification of the nodal structure of the model. Appropriate assumptions and constraints are then established, leading to the derivation of governing equations tailored to the model. Finally, the model is validated to ensure its accuracy and reliability. This methodology is applied to develop models for various components of a typical PWR plant, with a particular emphasis on the significance of assumptions and the derivation method for governing equations, which are explored in greater detail.

3.3 Assumptions

3.3.1 Mass and Energy conservation Principal

All the model developed and used in this study was based on the conservation of mass and energy in a closed system principle. The conservation of mass principle is based on the idea that total mass inflow in a system will be balanced by mass outflow. Mathematically,

$$\dot{M}_{in} - \dot{M}_{out} = 0 \tag{3.1}$$

where, $\dot{M}_{in}$ total mass flow rate into the system and $\dot{M}_{out}$ total mass flow rate of the system [15]. This balance equation presupposes the absence of mass generation within the system. Regarding the reactor dynamics' governing equations, although continuous fission results in a loss of mass over time, this mass loss is considered negligible for short-duration simulations and is therefore disregarded in this study.

The conservation of energy principle states that the sum of the energy inflow and energy generation within the system will be equal to the total energy outflow. Mathematically,

$$\sum_{system} (\dot{E}_{\text{in}} + \dot{E}_{\text{gen}}) = \sum_{system} \dot{E}_{\text{out}} \tag{3.2}$$

where, $\dot{E}_{\text{in}}$ represents the rate of energy inflow into the system, $\dot{E}_{\text{out}}$ signifies the rate of energy outflow from the system, and $\dot{E}_{\text{gen}}$ denotes the rate of energy generation within the system.

3.3.2 Fluid properties

It's very necessary to update the coolant properties on the fly of the simulation based on the instantaneous system state properties like temperature, pressure, and enthalpy. Most of the previous study on dynamic modelling of the reactor and other systems that involve circulating and heating the liquid presumes that the relationship is linear between them just to make the model simpler. Figure 3.2 shows a comparison of linear assumption vs actual data.

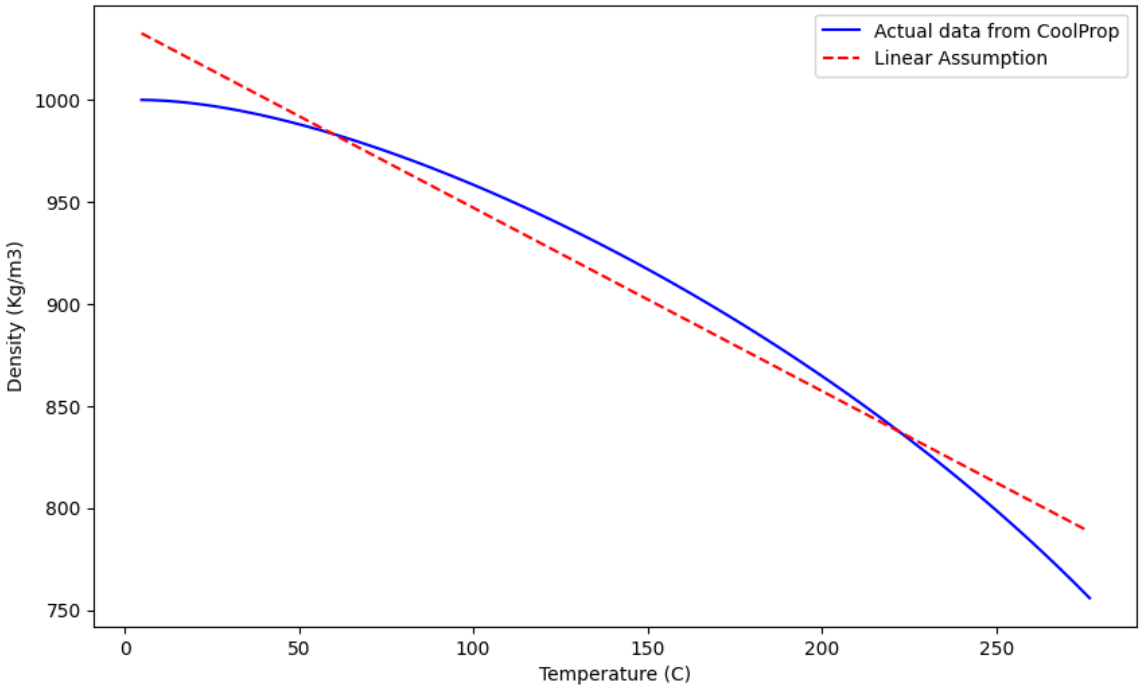

Figure 3.2: Saturated water temperature vs density

That's why to avoid any error such as the assumption of the linear relation between fluid properties, CoolProp was used. [16].

3.3.3 Pump Characteristics

The dynamics of reactor primary coolant pumps and other feedwater pumps are contingent upon their characteristic curves, as provided by the pump designers [15]. These curves delineate the relationship between the flow rate, denoted by $\dot{Q}$, and the developed head. This relationship can be mathematically represented as follows:

$$H(\dot{Q}) = \sum_{i=1}^{n} A_i \dot{Q}^i \tag{3.3}$$

where A_i is the i^{th} polynomial coefficient. The determination of A_i is essential to conduct the simulations. To this end, a program named `pump_const_determination.py` has been developed to ascertain the polynomial coefficients of the pump's characteristic curve. Figure **??** illustrates the pump characteristic data alongside the interpolated function.

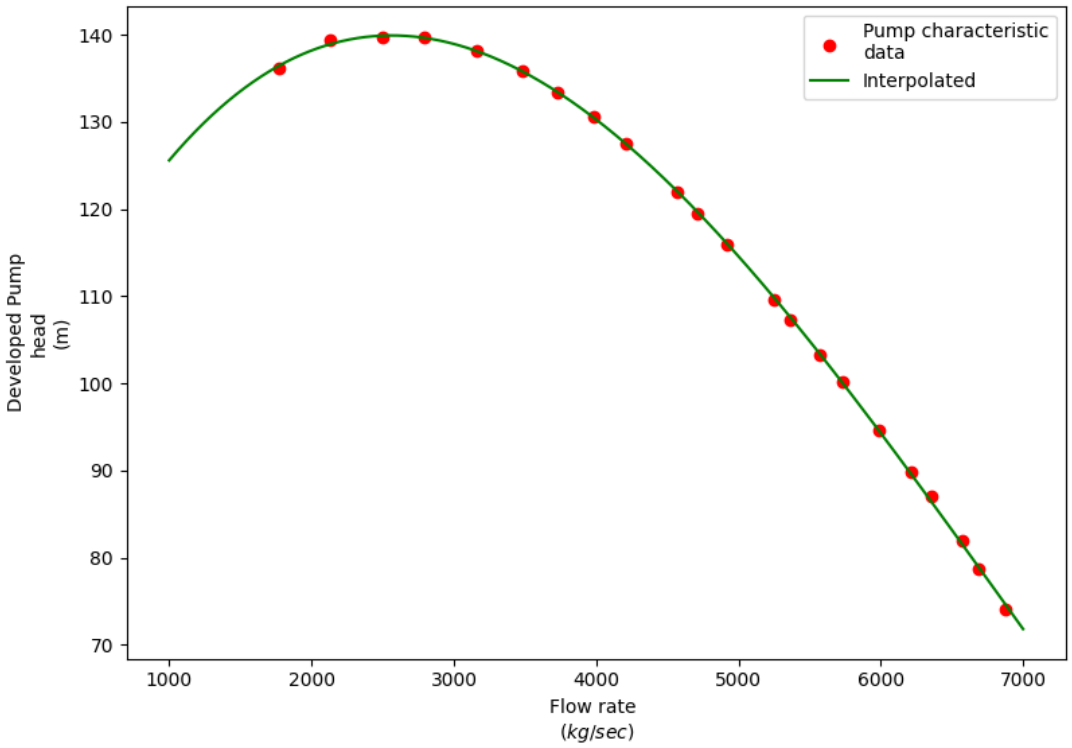

Figure 3.3: Pump flow rate vs head curve

3.3.4 Determination of experimental constants

There are some equations in turbine and condenser which rely on the experimentation of the system. For example, the low-pressure turbine's inlet flow rate is determined by an equation given by,

$$W_{lp1} = K_{clp}\sqrt{P_r \rho_r} \tag{3.4}$$

where,W_{lp}=Low pressure turbine inlet flow rate, P_r= reheater pressure, ρ_r=reheater density and K_{clp} a constant determined at the initial period of the startup [9]. Calculation of these constants needs to be from other existing literature or the designer manual of that component as they vary from power plant system to system. This study addresses the problem by manually calculating the values from the existing literature. In this programmed developed API the variable `K_lp1` under `secondary_loop_system` class is public, so if needed the user can change the preassigned value according to their model or adjust to their calculation.

3.4 Governing equation formation

3.4.1 Integrator

Most of the system governing equations are non-linear differential equations. These equations can be solved in two ways. Either by formatting a system matrix that is expressed as,

$$A\dot{\vec{X}}^T = \vec{b} \tag{3.5}$$

where A is the system matrix in which elements are calculated from the derived system of linear equations, $\dot{\vec{X}}$ is the state space variable and $\vec{b}$ is the system constant column vector or column matrix. This linear system can be solved either using iterative methods like Jacobi, Gauss-Seidel and Successive Over Relaxation (SOR) method or via direct inverse solution [17].

$$\dot{\vec{X}} = A^{-1}\vec{b} \tag{3.6}$$

Another method is expressing the time derivative of i^{th} for every single state space variable as a function of that system's state space variable. Mathematically,

$$\frac{d}{dt}\vec{X} = f(\vec{X}_i, ..., \vec{X}_n) \tag{3.7}$$

Equation 3.7 can be solved using an integration scheme that can do transient integration like Euler's integration method, which is mainly used in the study [17]. A program `integrator.py` has been developed to solve the transient integration problems in that study. Figure 3.4 shows a comparison of the exact solution and the solution obtained using the program `integrator.py` for a differential equation within the interval $[-\pi, \pi]$ given by

$$\frac{d}{dx}f(x) = \cos x \tag{3.8}$$

The exact solution of the equation 3.8 is $f(x) = \sin x$. More about the integration scheme and algorithms are discussed in Chapter 5.

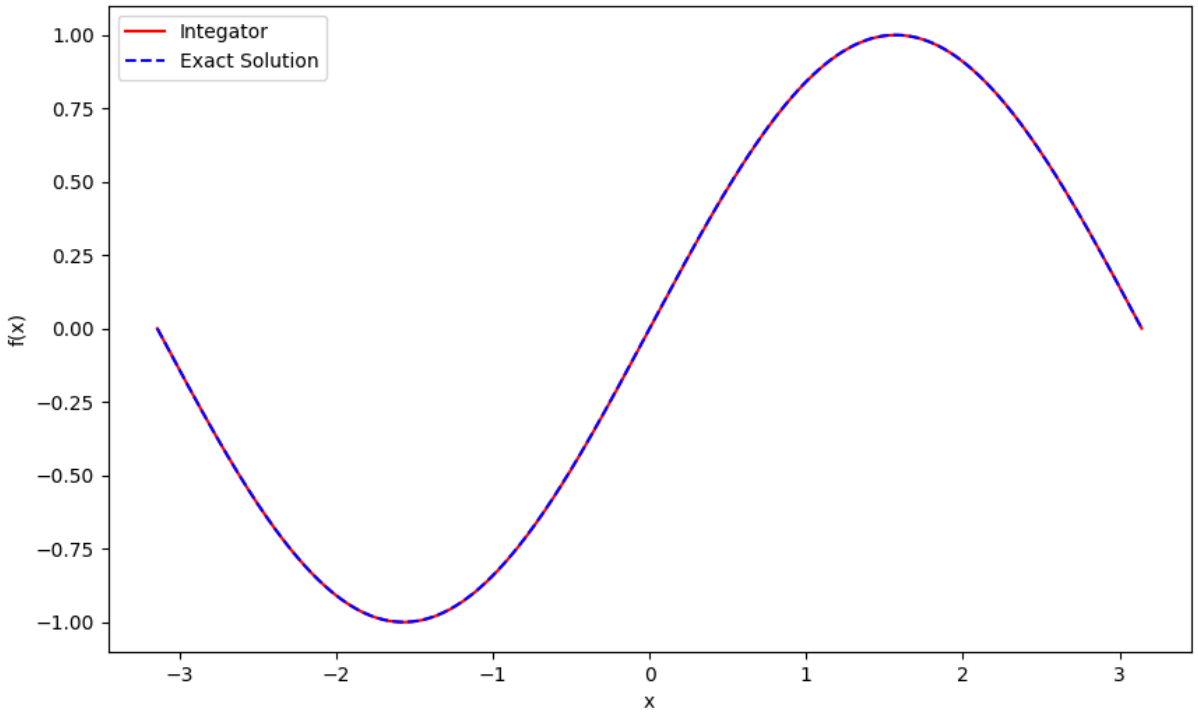

Figure 3.4: A comparison of exact and integrator's solution.

3.4.2 Non-linearity of the models

Most of the governing equations developed in that study are non-linear, so they have sensitivity to the initial conditions in the first place where the model `class` is defined. For a better realistic result, the initial should be close to the experimental result otherwise the program will predict a whole new equilibrium or non-equilibrium state[17]. Figure 3.5 portrays the sensitivity of the initial conditions.

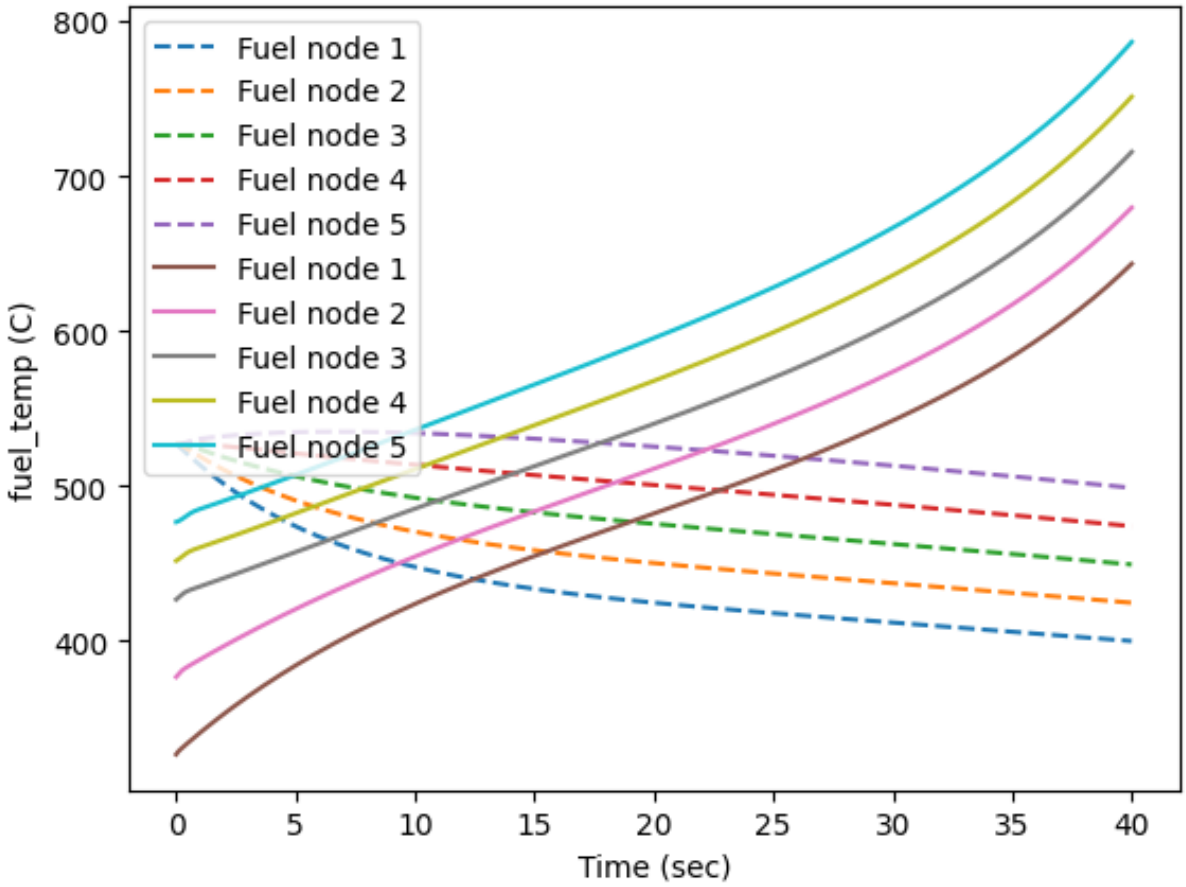

Figure 3.5: Sensitivity on the initial conditions

`--` `Correct initial condition` `-` `Ambigious initial condition`

Chapter 4

Mathematical Model Development of Power Plant Components

This chapter is dedicated to the development of the mathematical model of nuclear power plant components of generalized pressurized water or heavy water reactors. This chapter encompasses mathematical representations of the reactor core, steam generator, pressurizer, and the reactor primary coolant pump. The process of developing these models is elaborated upon in Chapter 3.

4.1 Reactor Core

The 4.1 shows a typical PWR reactor core where primary coolant flows in the core through the nozzles about the top of the reactor core it circulates throughout the core and extracts heat from the fuel rods. The water exits through the steam outlet of the reactor pressure vessel [4]. The main flow pattern of the water in the core is, Reactor primary coolant pump → Cold leg piping node → Lower plenum of the reactor → Core→ Upper plenum of the reactor core → Hot leg of the reactor.

4.1.1 Reactor Model

The main model of the reactor used here has the following two parts:

1. Neutronic portion
2. Thermal-hydraulic portion

The neutronic model will deal with the thermal energy production due to fission and the reactivity feedback due to the changes in the operational and core environmental variables. The thermal-hydraulic model will deal with the heat extraction from the fuel rods, heat and temperature distribution inside and the pressure development due to the

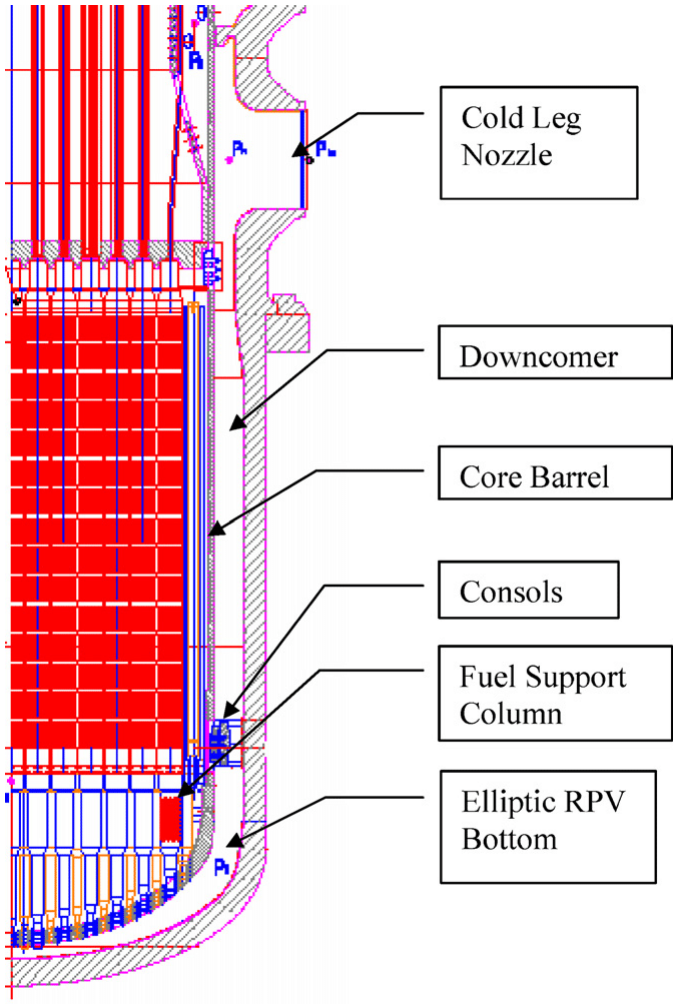

Figure 4.1: Main components of VVER-1200 reactor

heating of the coolant. The thermal-hydraulic model will work in a coupled manner by generating transient feedback for the neutronic model [18].

The reactor core model is developed by following a lumped approach where the core is divided into five parts for better insights into temperature distributional observation [4]. Each section contains a fuel fuel lump and two moderator or coolant lumps. There are a total of five fuel lumps and ten coolant or moderator lumps. The 4.2 portrays the detailed geometry of the model.

Neutronic Model

The neutronic model is based on the neutron point kinetics equation where a total of six groups of neutron precursors are considered. This study presumes that the average neutron density is proportional to the power ratio [18]. So the point kinetic equation becomes,

$$\frac{d(\frac{P}{P_0})}{dt} = \frac{(\rho - \beta_t)}{\Lambda}\frac{P}{P_0} + \lambda C \tag{4.1}$$

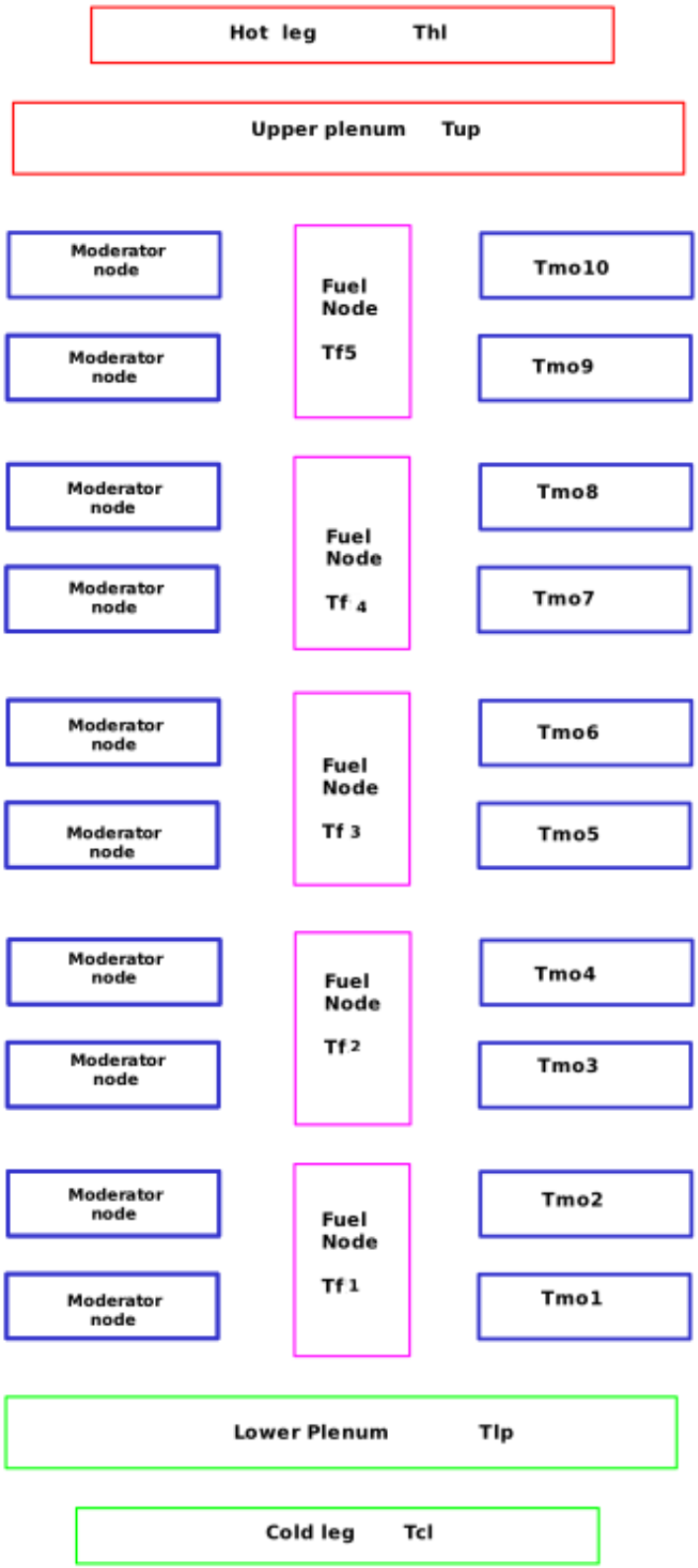

Figure 4.2: Diagram illustrating the schematic representation of the reactor model.

$$\frac{dC}{dt} = \frac{\beta_t \frac{P}{P_0}}{\Lambda} - \lambda C \tag{4.2}$$

where,

$$\lambda = \sum_{i=1}^{n=6} \frac{\beta_t}{\frac{\beta_i}{\lambda_i}}$$

$$\beta_t = \sum_{i=1}^{n=6} \beta_i$$

.

The reactivity feedback due to the Doppler effect, negative moderator temperature and positive coolant pressure is calculated as,

$$\rho = \rho_{external} + \alpha_f \delta T_f + \alpha_c \delta T_c + \alpha_p \delta P \tag{4.3}$$

Thermal hydraulic Model

The reactor core has 10 moderator lumps and 5 fuel lumps. The thermal-hydraulic model is based on Mann's model for reactor heat transfer [5]. The governing equations are,

First Node:

$$\frac{dT_{f1}}{dt} = \frac{f_r P}{m_{fuel} C_{fuel}} - \frac{(UA)_{fc}(T_{f1} - T_{mod1})}{m_{fuel} C_{fuel}} \tag{4.4}$$

$$\frac{dT_{mod1}}{dt} = \frac{(1-f_r)P}{m_{coolant} C_{coolant}} + \frac{(UA)_{fc}(T_{f1} - T_{mod1})}{m_{coolant} C_{coolant}} + \frac{2(T_{lp} - T_{mod1})\dot{m}}{m_{coolant}} \tag{4.5}$$

$$\frac{dT_{mod2}}{dt} = \frac{(1-f_r)P}{m_{coolant} C_{coolant}} + \frac{(UA)_{fc}(T_{f1} - T_{mod1})}{m_{coolant} C_{coolant}} + \frac{2(T_{mod1} - T_{mod2})\dot{m}}{m_{coolant}} \tag{4.6}$$

Second Node:

$$\frac{dT_{f2}}{dt} = \frac{f_r P}{m_{fuel} C_{fuel}} - \frac{(UA)_{fc}(T_{f2} - T_{mod3})}{m_{fuel} C_{fuel}} \tag{4.7}$$

$$\frac{dT_{mod3}}{dt} = \frac{(1-f_r)P}{m_{coolant} C_{coolant}} + \frac{(UA)_{fc}(T_{f2} - T_{mod3})}{m_{coolant} C_{coolant}} + \frac{2(T_{mod2} - T_{mod3})\dot{m}}{m_{coolant}} \tag{4.8}$$

$$\frac{dT_{mod4}}{dt} = \frac{(1-f_r)P}{m_{coolant} C_{coolant}} + \frac{(UA)_{fc}(T_{f2} - T_{mod3})}{m_{coolant} C_{coolant}} + \frac{2(T_{mod3} - T_{mod4})\dot{m}}{m_{coolant}} \tag{4.9}$$

Third Node:

$$\frac{dT_{f3}}{dt} = \frac{f_r P}{m_{fuel} C_{fuel}} - \frac{(UA)_{fc}(T_{f3} - T_{mod5})}{m_{fuel} C_{fuel}} \tag{4.10}$$

$$\frac{dT_{mod5}}{dt} = \frac{(1-f_r)P}{m_{coolant} C_{coolant}} + \frac{(UA)_{fc}(T_{f3} - T_{mod5})}{m_{coolant} C_{coolant}} + \frac{2(T_{mod4} - T_{mod5})\dot{m}}{m_{coolant}} \tag{4.11}$$

$$\frac{dT_{mod6}}{dt} = \frac{(1-f_r)P}{m_{coolant} C_{coolant}} + \frac{(UA)_{fc}(T_{f3} - T_{mod5})}{m_{coolant} C_{coolant}} + \frac{2(T_{mod5} - T_{mod6})\dot{m}}{m_{coolant}} \tag{4.12}$$

Fourth Node:

$$\frac{dT_{f4}}{dt} = \frac{f_r P}{m_{fuel} C_{fuel}} - \frac{(UA)_{fc}(T_{f4} - T_{mod7})}{m_{fuel} C_{fuel}} \tag{4.13}$$

$$\frac{dT_{mod7}}{dt} = \frac{(1-f_r)P}{m_{coolant} C_{coolant}} + \frac{(UA)_{fc}(T_{f4} - T_{mod7})}{m_{coolant} C_{coolant}} + \frac{2(T_{mod6} - T_{mod7})\dot{m}}{m_{coolant}} \tag{4.14}$$

$$\frac{dT_{mod8}}{dt} = \frac{(1-f_r)P}{m_{coolant} C_{coolant}} + \frac{(UA)_{fc}(T_{f4} - T_{mod7})}{m_{coolant} C_{coolant}} + \frac{2(T_{mod7} - T_{mod8})\dot{m}}{m_{coolant}} \tag{4.15}$$

Fifth Node:

$$\frac{dT_{f5}}{dt} = \frac{f_r P}{m_{fuel} C_{fuel}} - \frac{(UA)_{fc}(T_{f5} - T_{mod9})}{m_{fuel} C_{fuel}} \tag{4.16}$$

$$\frac{dT_{mod9}}{dt} = \frac{(1-f_r)P}{m_{coolant} C_{coolant}} + \frac{(UA)_{fc}(T_{f5} - T_{mod9})}{m_{coolant} C_{coolant}} + \frac{2(T_{mod8} - T_{mod9})\dot{m}}{m_{coolant}} \tag{4.17}$$

$$\frac{dT_{mod10}}{dt} = \frac{(1-f_r)P}{m_{coolant} C_{coolant}} + \frac{(UA)_{fc}(T_{f5} - T_{mod9})}{m_{coolant} C_{coolant}} + \frac{2(T_{mod9} - T_{mod10})\dot{m}}{m_{coolant}} \tag{4.18}$$

Cold Leg:

$$\frac{dT_{cl}}{dt} = \frac{\dot{m}(T_{RCP} - T_{cl})}{m_{cl}} \tag{4.19}$$

Lower Plenum:

$$\frac{dT_{lp}}{dt} = \frac{\dot{m}(T_{cl} - T_{lp})}{m_{lp}} \tag{4.20}$$

Upper Plenum:

$$\frac{dT_{up}}{dt} = \frac{\dot{m}(T_{mod10} - T_{up})}{m_{up}} \tag{4.21}$$

Hot Leg:

$$\frac{dT_{hl}}{dt} = \frac{\dot{m}(T_{up} - T_{hl})}{m_{hl}} \tag{4.22}$$

The list of the variables is given the table below 4.1.

Table 4.1: Reactor Model Variables

Variable	Definition
A_{eff}	Effective area heat transfer between fuel rod and moderator
C	Precursor concentration
C_{p_c}	Specific heat of the moderator
C_{p_f}	Heat capacity of the fuel elements
f_r	Fission power factor
U	Average overall heat transfer coefficient
$\dot{M}$	Flow rate
M_c	Moderator mass in moderator plenum.
M_{cl}	Mass in the cold leg
M_f	Fuel element mass in one node
M_{hl}	Mass in the hot leg
M_{lp}	Mass in the lower plenum
M_{co}	Coolant lump mass
M_{up}	Mass in the upper plenum
P	Reactor core power
T_cl	Temperature in the primary coolant's cold leg.
T_{f1-5}	Fuel temperatures in lumps (1-5)
T_{hl}	Temperature in the primary coolant's hot leg.
T_{lp}	Temperature in the primary coolant's cold leg.
$T_{modl-10}$	Moderator temperatures in lumps (1-10)
T_{up}	Temperature in the primary coolant's upper plenum.
T_{RCP}	Outlet temperature of the primary water leaving reactor primary coolant pump
α_c	Temperature reactivity feedback of the moderator
α_p	Pressure reactivity feedback of the moderator
α_f	Temperatre reactivity feed back of the fuel
β_t	Total delayed neutron group fraction
λ	Average of six group decay constant
Λ	prompt neutron life time.
ρ	Total reactivity
ρ_{ex}	External reactivity due to control rod movement

4.2 Pressurizer

A pressurizer is an essential component in a Pressurized Water Reactor (PWR) nuclear power plant, primarily responsible for maintaining the core pressure. It operates by man-

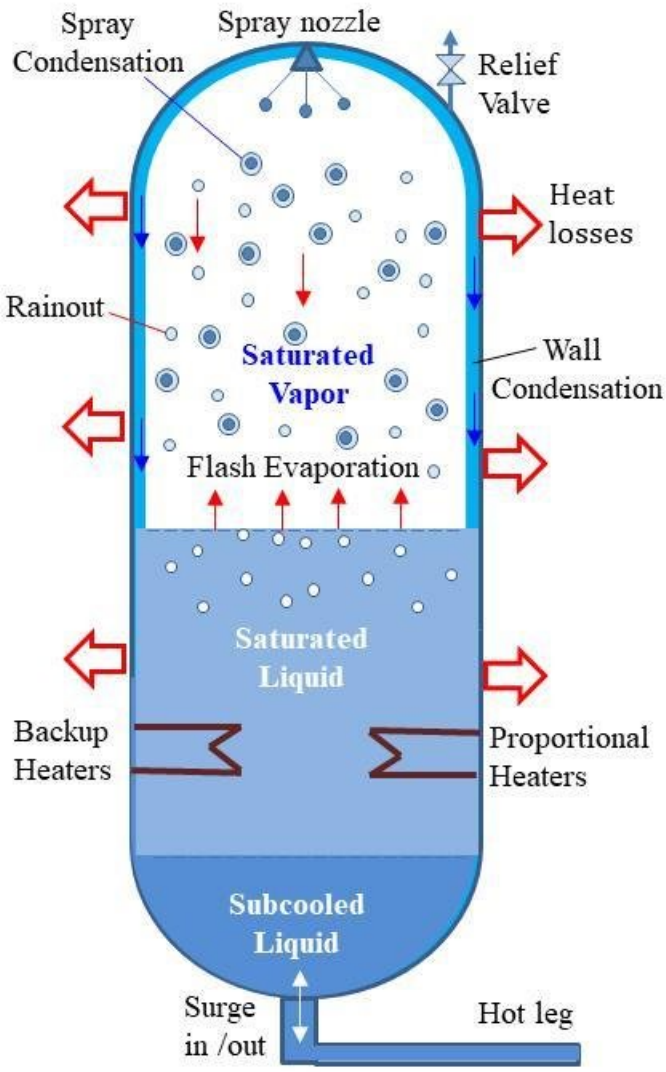

Figure 4.3: Schematic diagram of a PWR pressurizer [18]

aging changes in the reactor coolant volume that result from temperature fluctuations in the primary circuit. The pressurizer integrates a surge flow mechanism from the reactor's hot leg and a spray system from the cold leg. Additionally, it contains an internal heater to assist in pressure regulation by evaporating coolant when necessary. The design in Figure 4.3 ensures a stable mixture of liquid and vapour inside, always maintaining saturation conditions that align with the primary coolant pressure.

In the operation of the pressurizer, several assumptions are made for simplicity and efficiency. We assume that the surge flow blends completely with the liquid within the pressurizer, and there is no noteworthy condensation or heat loss at the vessel's walls and liquid interfaces. The initial values of the spray flow rate and heater output are considered non-impactful to the overall function. Moreover, the model accounts for steam compressibility as a function of the thermodynamic properties of water and steam, providing a pragmatic approach to represent pressure variations in the system [8].

4.2.1 Governing Equations

The governing equation of pressurizer dynamics are previously studied [8] [5] and can be given as,

$$\frac{dl_w}{dt} = \frac{1}{\rho_s A_{pressurizer}}((A_{pressrizer}(l-l_w)K_{2p} - \frac{C_{2p}}{C_{1p}})\frac{dP_p}{dt} + \frac{1}{C_{1p}^2}(C_{2p}\frac{dP_p}{dt} - \dot{m_{sur}} - \dot{m_{spr}}) + \frac{\dot{m_{sur}}}{C_{1p}}) \tag{4.23}$$

where,

$$C_{1p} = \frac{\rho_w}{\rho_s}$$

$$C_{2p} = A_{pressurizer}(l - l_w)\frac{\rho_w}{\rho_s}\frac{\delta\rho_s}{\delta P_p} + A_{pressrizer}l_w\frac{\delta\rho_w}{\delta P_p}$$

$$K_{2p} = \frac{\delta\rho_s}{\delta P_p}$$

$$\frac{dP_p}{dt} = \frac{Q_{heater} - \dot{m_{sur}}(\frac{P_p\nu_s}{J_pC_{1p}} + \frac{h_w}{C_{1p}}) + (\dot{m_{spr}}(h_{spr} - h_w + \frac{h_{\bar{w}}}{C_{1p}} + \frac{P_p\nu_w}{J_pC_{1p}}))}{m_w(K_{3p} + \frac{K_{4p}P_p}{J_p}) + \frac{m_sK_4P_p}{J_p} - \frac{V_w}{J_p} + \frac{C_{2p}}{C_{1p}}(h_{\bar{w}} + \frac{P_p\nu_s}{J_p})} \tag{4.24}$$

where,

$$k_{1p} = \frac{\delta\rho_w}{\delta P_p}$$

$$k_{3p} = \frac{\delta h_w}{\delta P_p}$$

$$k_{4p} = \frac{\delta\nu_w}{\delta P_p}$$

$\dot{m_{sur}}$ can be given as,

$$\dot{m_{sur}} = \sum_{j=1}^{j=N} V_j\nu_j\frac{dT_j}{dt} \tag{4.25}$$

The basic properties of the water and steam can be accessed by using CoolProp [16]

Table 4.2: Pressurizer Model Variables.

Variable	Definition
A_{pr}	Area of the pressurizer cross section
h_f	Saturated enthalpy of water
h_{fg}	Latent heat of the vaporization
h_{sp}	enthalpy at the spray nozzle
J	Heat conversion factor
K_{p1-p6}	$\frac{\partial u_t}{\partial p_r}, \frac{\partial v_w}{\partial p_r}, \frac{\partial v_s}{\partial p_r}, \frac{\partial h_f}{\partial p_r}, \frac{\partial h_{fg}}{\partial p_r}, \frac{\partial h_{sp}}{\partial p_r}$
L	Pressurizer height
L_w	Water level
M_s	Mass of steam
M_w	Mass of water
P_{pr}	Pressure
m'_{co}	Evaporation flow rate
m'_{sp}	Spray flow rate
m_{sr}	In/OutSurge flow rate
V_w	Water volume
ρ_s	Steam density
ρ_w	Water density

4.3 Steam Generator

Two common steam generators (SGs) are used in PWRs:

(1) Re-circulation (U-tube steam generators)
(2) Once-through (helical steam generators)

In a U-tube SG, heated coolant at high pressure from the reactor core enters at the bottom and follows an upward and then downward path through several thousand inverted U-shaped tubes[4, 19]. In a once-through SG, which usually employs a counter-flow heat exchanger, the primary coolant enters at the top and flows downward through tubes and leaves the SG at the bottom. With this design, a dry vapour or a few degrees of superheated steam can be produced [5].

In this study, only the U-tube steam generator model is developed. The mathematical model of the U-tube steam generator is based on a previous work where fluid properties were presumed to have a linear relationship with each other. The model developed by Ali

in his PhD thesis was very detailed as he varied the lumping to find the optimum design and model to predict the UTSG behaviour with an experimental benchmark [6].

4.3.1 U-tube steam generator

The design of the U-tube steam generator model was based on model D developed by Ali but this study presumes no linear relation between fluid properties to calculate their partial derivatives. All the coolant and steam properties were either calculated or exported from CoolProp's python API[16]. Figure: 4.4 4.4 is the model of the U-tube steam generator of the original study.

Governing Equations of UTSG:

Primary Side Equations:

Inlet Plenum:

$$\frac{dT_{pi}}{dt} = \frac{\dot{m_{pi}}(T_{hl} - T_{pi})}{m_{pi}} \tag{4.26}$$

Primary Lump (PRL1):

$$\frac{dL_{s1}}{dt} = \frac{\dot{m_{pi}} - \dot{m_{p1}}}{\rho_p A_p} \tag{4.27}$$

$$\frac{dT_{p1}}{dt} = \frac{\dot{m_p} C_{p1} T_{pi} - C_{p1} T_{p1}(\dot{m_{pi}} - (\dot{m_{pi}} - \dot{m_{p1}})) - U_{pm} P_{r1} L_{s1}(T_{p1} - T_{m1})}{\rho_p A_p C_{p1} L_{s1}} - T_{p1} \frac{\dot{m_{pi}} - \dot{m_{p1}}}{\rho_p A_p L_{s1}} \tag{4.28}$$

Primary Lump (PRL2):

$$\frac{dL_{s2}}{dt} = \frac{\dot{m_{p1}} - \dot{m_{p2}}}{\rho_p A_p} \tag{4.29}$$

but as $L_{s1} + L_{s2} = L$ the equation 4.29 can also be written as

$$\frac{dL_{s2}}{dt} = -\frac{dL_{s1}}{dt}$$

$$\frac{dT_{p2}}{dt} = \frac{\dot{m_{pi}}(T_{p1} - T_{p2})}{m_{p2}} - \frac{U_{pm} P_{r1} L_{s2}(T_{p2} - T_{m2})}{m_{p2} C_{p1}} - \frac{(T_{p1} - T_{p2})(\dot{m_{pi}} - \dot{m_{p1}})}{L_{s2} \rho_p A_p} \tag{4.30}$$

Primary Lump (PRL3):

$$\dot{m_{p3}} = 2\dot{m_{pi}} - \dot{m_{p1}} \tag{4.31}$$

$$\frac{dT_{p3}}{dt} = \frac{\dot{m_{pi}}(T_{p2} - T_{p3})}{m_{p3}} - \frac{U_{pm} P_{r1} L_{s2}(T_{p3} - T_{m3})}{m_{p3} C_{p1}} \tag{4.32}$$

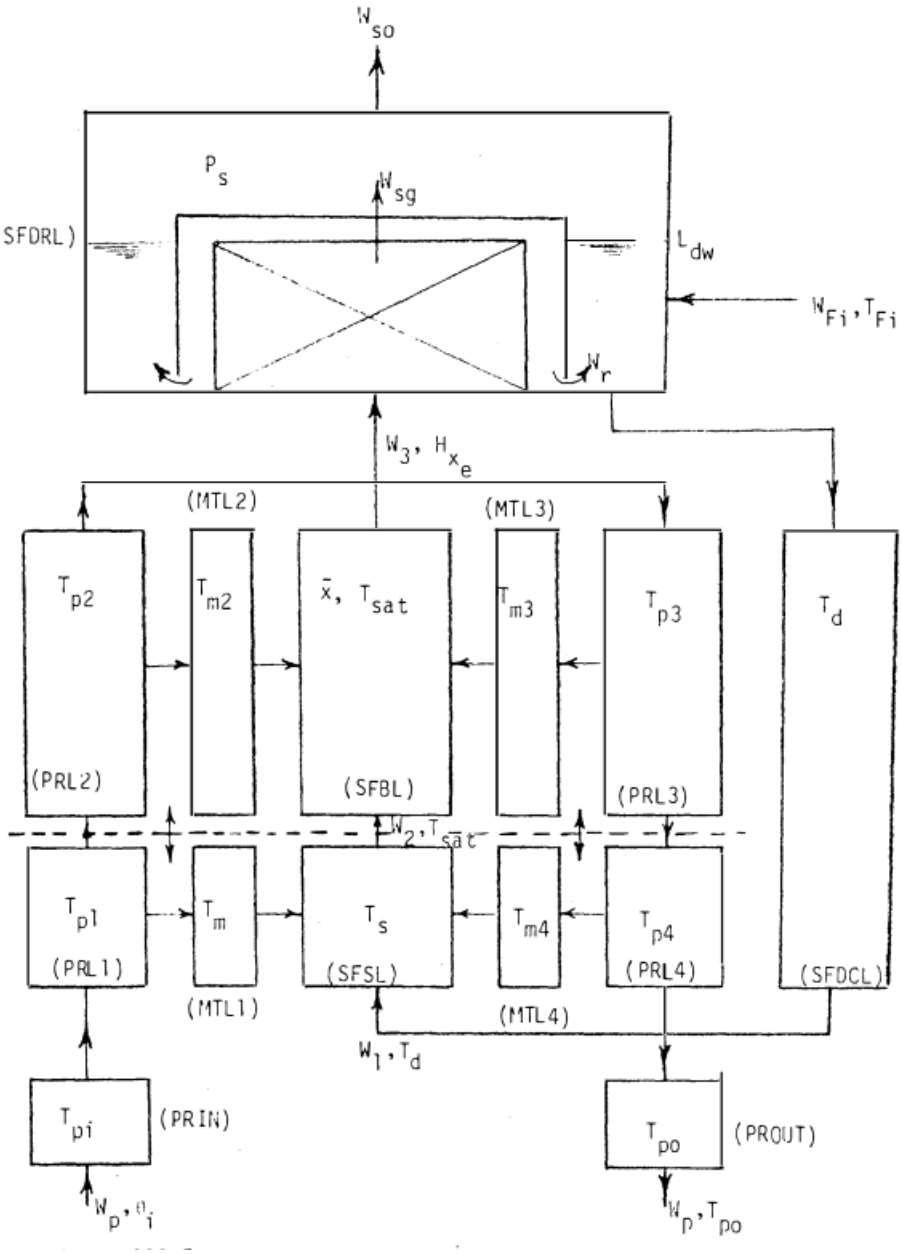

Figure 4.4: Schematic diagram of the UTSG model D [6].

Primary Lump (PRL4):

$$\frac{dT_{p4}}{dt} = \frac{\dot{m_{pi}}(T_{p3} - T_{p4})}{m_{p4}} - \frac{U_{pm}P_{r1}L_{s1}(T_{p4} - T_{m4})}{m_{p4}C_{p1}} - \frac{(T_{p4} - T_{p3})(\dot{m_{pi}} - \dot{m_{p1}})}{\rho_p A_p L_{s1}} \tag{4.33}$$

also,

$$m_{p4} = m_{p1}$$

$$m_{p2} = m_{p3}$$

Outlet Plenum:

$$\frac{dT_{po}}{dt} = \frac{\dot{m_{pi}}(T_{p4} - T_{po})}{m_{pi}} \tag{4.34}$$

Metal Lump:

Tube Metal Lump1:

$$\frac{dT_{m1}}{dt} = \frac{(U_{pm}P_{r1}L_{s1}(T_{p1} - T_{m1}) - U_{ms1}P_{r2}L_{s1}(T_{m1} - T_{s1}))}{m_{m1}C_m} - \frac{(T_{m1} - T_{m2})(\dot{m_{pi}} - \dot{m_{p1}})}{\rho_p A_p L_{s1}} \tag{4.35}$$

Tube Metal Lump2:

$$\frac{dT_{m2}}{dt} = \frac{U_{pm}P_{r1}L_{s2}}{m_{m2}C_m}T_{p2} - \frac{U_{pm}P_{r1}L_{s2} + U_{ms2}P_{r2}L_{s2}}{m_{m2}C_{m2}}T_{m2} - \frac{U_{ms2}P_{r2}L_{s2}}{m_{m2}C_m}T_{sat} + \frac{T_{m2} - T_{m1}}{2L_{s2}}\frac{dL_{s1}}{dt} \tag{4.36}$$

where,

$$\frac{dL_{s1}}{dt} = \frac{\dot{m_{pi}} - \dot{m_{p1}}}{\rho_p A_p}$$

Tube Metal Lump3:

$$\frac{dT_{m3}}{dt} = \frac{U_{pm}P_{r1}L_{s2}}{m_{m2}C_m}T_{p3} - \frac{U_{pm}P_{r1}L_{s2} + U_{ms2}P_{r2}L_{s2}}{m_{m2}C_{m2}}T_{m3} - \frac{U_{ms2}P_{r2}L_{s2}}{m_{m2}C_m}T_{sat} + \frac{T_{m3} - T_{m4}}{2L_{s2}}\frac{dL_{s1}}{dt} \tag{4.37}$$

where,

$$\frac{dL_{s1}}{dt} = \frac{\dot{m_{pi}} - \dot{m_{p1}}}{\rho_p A_p}$$

Tube Metal Lump4:

$$\frac{dT_{m4}}{dt} = \frac{U_{pm}P_{r1}L_{s1}}{m_{m1}C_m}T_{p4} - \frac{U_{pm}P_{r1}L_{s1} + U_{ms2}P_{r2}L_{s1}}{m_{m1}C_{m2}}T_{m4} - \frac{U_{ms2}P_{r2}L_{s1}}{m_{m1}C_m}T_{s1} - \frac{T_{m3} - T_{m4}}{2L_{s1}}\frac{dL_{s1}}{dt} \tag{4.38}$$

where,

$$\frac{dL_{s1}}{dt} = \frac{\dot{m_{pi}} - \dot{m_{p1}}}{\rho_p A_p}$$

Secondary Region:

Secondary fluid sub-cool liquid(SFSL):

$$\frac{dT_{sat}}{dt} = \frac{U_{ms1}P_{r2}L_{s1}(T_{m1}+T_{m2}-2T_{s1}) - \dot{m}_1 C_{p2}T_d - \dot{m}_2 C_{p2}T_{sat} - \rho_{s1}A_{fs}C_p T_{s1}\frac{dL_{s1}}{dt} - \frac{1}{2}m_{s1}C_p\frac{dT_d}{dt}}{\rho_{s1}A_{fs}C_p L_{s1}} \tag{4.39}$$

where,

$$T_{s1} = \frac{T_d + T_{sat}}{2}$$

$$\frac{dL_{s1}}{dt} = \frac{\dot{m}_1 - \dot{m}_2}{\rho_p A_p}$$

the value $\dot{m}_2$ is calculated from the $\frac{dL_{s1}}{dt}$ derivative given that $\dot{m}_1$ is known already.

Secondary fluid boiling liquid(SFBL):

$$\frac{d\rho_b}{dt} = \frac{(\dot{m}_2 - \dot{m}_3)}{A_{fs}L_{s2}} + \frac{\rho_b}{L_{s2}}\frac{dL_{s1}}{dt} \tag{4.40}$$

$$\frac{dh_b}{dt} = \frac{U_{ms2}P_{r2}L_{s2}(T_{m2}-T_{sat}) + U_{ms2}P_{r2}L_{s2}(T_{m3}-T_{sat}) + \dot{m}_2 h_f - \dot{m}_3 h_{x_e} + \rho_b A_{fs}h_b\frac{dL_{s1}}{dt}}{\rho_b A_{fs}L_{s2}} \tag{4.41}$$

where,

$$\frac{dL_{s1}}{dt} = \frac{\dot{m_{pi}} - \dot{m_{p1}}}{\rho_p A_p}$$

$$h_{x_e} = h_f + x_e(h_g - h_f)$$

for the calculation of h_{x_e} the command PropsSI from CoolProp's API [16] is used.

Drum equivalent secondary fluid region (SFDRL):

$$\frac{d\rho_r}{dt} = \frac{(\dot{m}_3 - \dot{m}_2)}{V_r} \tag{4.42}$$

Drum water volume:

$$\frac{dL_{dw}}{dt} = \frac{\dot{m_{fi}} - (1-x_e)\dot{m}_4 - \dot{m}_1}{\rho_{dw}A_{dw}L_{dw}} \tag{4.43}$$

$$\frac{dT_w}{dt} = \frac{1}{L_{dw}}\left(\frac{\dot{m_{fi}}T_{fi} - (1-x_e)\dot{m}_4 T_{sat} - \dot{m}_1 T_w}{\rho_{dw}A_{dw}} - T_w(\dot{m_{fi}} - (1-x_e)\dot{m}_4 - \dot{m}_1)\right) \tag{4.44}$$

$$\frac{d\rho_g}{dt} = \frac{x_e\dot{m}_4 - C_l P + \rho_g A_{dw}(\dot{m_{fi}} - (1-x_e)\dot{m}_4 - \dot{m}_1)}{V_r - A_{dw}L_{dw}} \tag{4.45}$$

Re circulation loop flow $\dot{m}_1$ is calculated as

$$\dot{m}_1 = C_1\sqrt{\rho_d(L_{dw}+L_d-L_{s1}) - \rho_r L_r}$$

here $C1 = \frac{1}{\sqrt{C_d}}$ and C_d is the friction loss coefficient in the down comer lump. The value of the C_d is normally provided by the designer of the steam generator.

Table 4.3: Steam Generator Model Variables

Variable	Definition
A_{fs}	Secondary flow area
A_{dw}	area of the drum water lump
C_l	friction factor in the down comer lump
C_t	Steam valve coefficient
C_m	Specific heat capacity of the metal tubes
$C_{p1,2}$	Specific heat capacity of the primary fluid and sub cooled region
h_b	boiling lump enthalpy
$h_{f,sg}$	Saturated and latent enthalpies
h_{ex}	Exit enthalpy of the boiling region
L	Length of the U-tubes
L_d	Down comer length
L_{dw}	Drum water level
$L_{s1,2}$	Sub cooled and boiling lengths
$M_{m1,2}$	Metal lump mass $(1, 2)$
M_{pi-4}	primary lump mass $(1 - 4)$
M_{pi}	inlet plenum mass
P	Pressure
$P_{r1,2}$	total inside and outside circumference of the U-tubes
$S_{ms1,2}$	Heat transfer areas in the sub cooled and boiling regions respectively
$S_{pm1,2}$	Heat transfer areas from the primary side to the U-tubes in nodes $(1, 2)$

4.4 Primary Coolant Pump:

The mathematical model of the reactor primary coolant pump (RCP) is based on work of Naghedolfeizi [4] and Li [20]. This model presumes that the pump inertia, I and power delivered, P_d is constant (in the program it will be user defined). The inlet of the RCP is just the exit of the steam generator and the outlet of the RCP supplies coolant to the cold leg of the reactor core. This model presumes that there is no coolant, piping loss [4] and t pump head is a polynomial function of RCP coolant flow rate [15]. A code has been developed to solve the pump characteristic curve provided by the designers to fit a third degree polynomial which express H as a function of $\dot{Q}$.

Governing Equation:

$$\frac{d\dot{Q}}{dt} = \frac{gA_{eff}((H_p - K\dot{Q}^2)}{L} \tag{4.46}$$

where,

$$H_p = \sum_{i=0}^{N} A_i (Q_i)^i$$

rotation is given as,

$$\frac{dN_p}{dt} = \frac{(P_d - \rho g Q H_p)}{4\pi^2 N_p I} \tag{4.47}$$

Table 4.4: Reactor Primary Coolant Pump Variables

Variable	**Definition**
A_i	Constant parameters of the pump characteristic equation
A_{cf}	Piping area
g	Gravitational acceleration
H_l	Total hydrodynamic head losses in the primary side
H_p	pump head
I	moment of inertia
L	Effective length of pipings in the primary loop
K	Friction factor
N_p	pump speed
Q_p	pump flow rate
P_d	Power (pump)
T_q	Torque (pump shaft)
T_h	torque (Hydrodynamic)
ρ	Density of water

Chapter 5

Algorithms and Data Management

The main goal of this study is to solve all the mathematical models governed by non-linear differential equations in a coupled manner. This suggests that every transient solution of the non-linear differential equations will have an impact on others' solutions at any instantaneous time. In this chapter, the managing structure of all these equations, the program development procedure and how the database will store the data generated by the program are discussed.

5.1 The Concept of Object Oriented Programming:

In the landscape of dynamic modelling, the adoption of Object-Oriented Programming (OOP) presents an elegant solution to represent complex, evolving systems. Python, recognized for its clarity and adaptability, serves as a formidable platform for the application of OOP principles in the domain of dynamic modelling. This exploration aims to elucidate the synergies between OOP and Python, employing a methodical approach to leverage abstraction, encapsulation, and polymorphism for the systematic representation of dynamic systems.

Inheritance, a core tenet of OOP, assumes a central role in this discourse, offering a mechanism to navigate the complexities inherent in dynamic modelling. Python's inherent support for fluid inheritance structures enables the creation of hierarchical models that mirror the evolving relationships and dependencies within dynamic systems. The discussion extends to polymorphism, a dynamic capability vital for adapting objects to varying contexts within dynamic systems. Python's dynamic typing and duck typing mechanisms exemplify the concept of polymorphism, permitting the modelling of diverse entities while preserving code flexibility and readability. This discourse endeavours to articulate the synergy between Python's OOP paradigm and dynamic modelling, delineating a systematic approach to represent, understand, and manipulate complex systems. The principles elucidated herein aim to provide a foundational understanding, empowering

practitioners to apply OOP in Python for effective dynamic modelling [21].

5.1.1 OOP for modelling

There are two ways which can be followed for any dynamic modelling.

1. Functional Programming.
2. Object-oriented Programming.

```
        def f(x):
        ... ... ...
        ... ... ...
        #fomula of the respective derivative function
        ... ... ...
        ... ... ...
        return value_of_the_time_derivative_of_state_variable
```

In this way, all the derivative functions will be global and one change in any state space variable will affect the other dependent variables. But it will make the feedback update system after a differential time, `dt` a bit complex to handle. It requires less computing memory given that data collection methods are simple and can easily be stored in a data management system. But using multiple components in the simulations for example two steam generators or multiple low-pressure feed water heaters becomes a tough job given that the same type of components are governed by the same physical laws, which only makes the program more repetitive. Naghedolfeizi [4] in his study used this method.

In object-oriented modelling, all the governing systems of a specific component will be under a class. The user will use the `det __init__():"` function of that specific class to declare a variable of that `component_type_class`. And all the governing equations will be under a specific type of `class`. The main formation will be,

```
    class A():
        def __init__(self,arguments):
            ......................
            ......................
        def f(self,arguments):
            ......................
            ......................

            return
               ↪ value_of_the_time_derivative_of_state_variable
               ↪
```

```
        def integrator(self,function,arguments_for_function,
            ↪ initial_condition,
                        differential_time_step):
            integral_value=function(arguments_for_function)*
                ↪ dt+initial_condition
            return inegral_value
```

In this way, the variables and time differential functions will be under a class which makes the whole simulation for the user easy to access and in a better control of the simulation. It will also help the user to simulate any conditions in any component. One of the major advantages that the OOP approach offers is the assurance of using multiple components of the same type. For example consecutive steam generator assays or a series of small-scale modular reactors.

5.2 Algorithms

One of the main targets of this study is to solve the system governing non-linear differential equations in a time domain. For the simulation of any operational and anomaly dynamics, the algorithm needs to be very precise and efficient as the program will handle a lot of data at any transient time. Besides the time loop will be integrating those equations as well. For simulating anomalies or operation of any instrument, the user will create conditionals within the loop. For example, driving control rods into the reactor core within the period $[t_1, t_2]$ will be like this,

```
    if t=>t1 and t<=t2:
        reactor_core.external_reactivity=
            ↪ desired_control_rod_worth
    else:
        reactor_core.external_reactivity =0
```

here the term, `reactor_core.external_reactivity` the external reactivity inserted by the control rods. The program starts by importing the `AZOG` (AZOG is the name of the developed program in this study) library and declaring the component class. Then, it prepares the database for tracking data and sets the initial conditions and time steps. Next, the simulation starts at time zero. The program then obtains the differential of the state space variables and calculates the solution of the differential equations in the time domain using transient fluid properties from CoolProp. The data is then written to the database and merged with the system. If there are no operational or anomalous conditions, the time step is increased. Otherwise, the program continues to write data to the database and compare it to the system until the simulation time is reached. Finally,

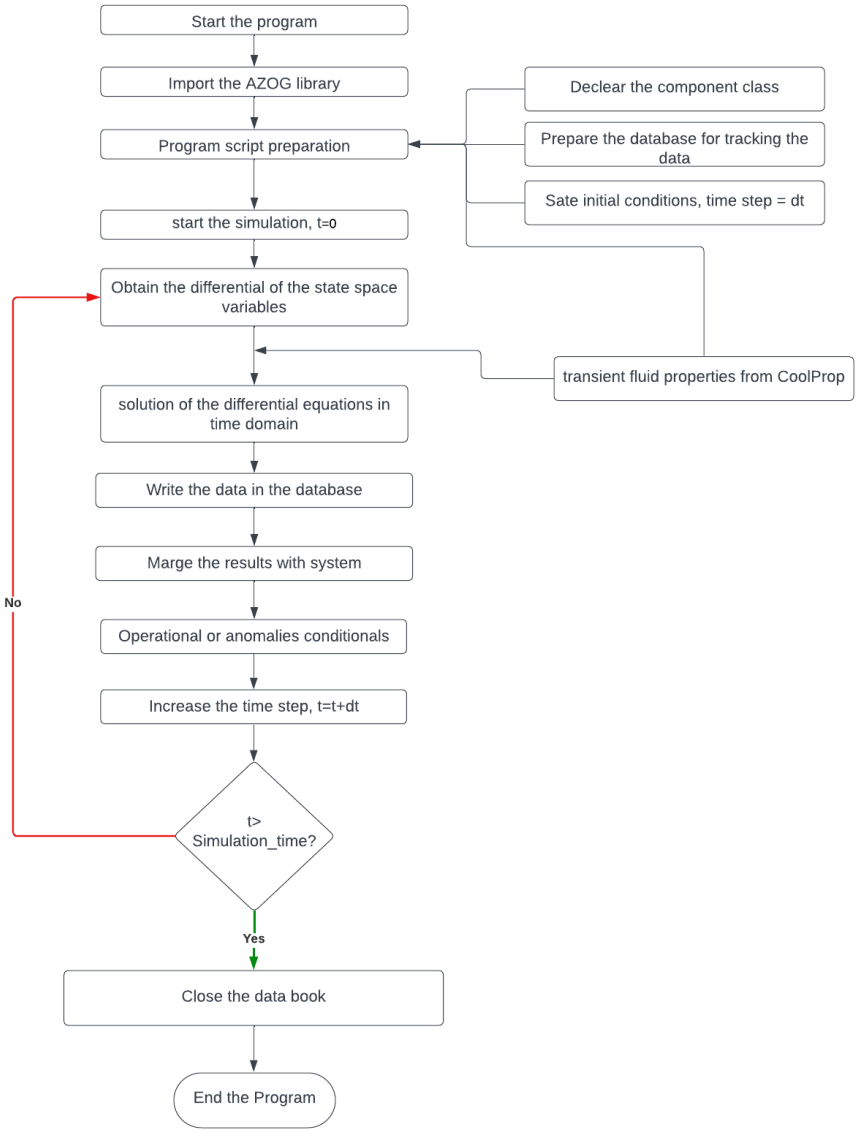

Figure 5.1: Program Flow chart

the program closes the data book and ends. The details algorithms of the program are given in the following flow chart 5.1. Also, a sudo code is given as a reference,

```
#SUDO CODE:

from AZOG.steam_generator import u_tube_steam_generator
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

UTSG=u_tube_steam_generator(primary_coolant_inlet_temperature
   ↪ = ... ,
    feed_water_inlet_temperature=.. ,
    drum_water_temp=... ,avg_sub_cool_temp=... ,
    feed_water_flow_rate=1000,
    PrimaryLumpTemperature=np.array([... , ... , ..., ...]),
    MetalLumpTemperature=np.array([... , ... , ..., ...]),
    Pressure=... ...)

Simulation_time=100000
dt=1
t=0
while t<Simulation_time:

    UTSG.Tpi=UTSG.integrator(UTSG.DTpi,argsforfunction=[...],
            intitial_cond=UTSG.Tpi,time_step=dt)
    UTSG.L_w=UTSG.integrator(UTSG.DLs1,argsforfunction=[],
            intitial_cond=UTSG.L_w,time_step=dt)
    UTSG.Tp1=UTSG.integrator(UTSG.DTp1,argsforfunction=[],
            intitial_cond=UTSG.Tp1,time_step=dt)
    UTSG.Tp2=UTSG.integrator(UTSG.DTp2,argsforfunction=[],
            intitial_cond=UTSG.Tp2,time_step=dt)
    UTSG.Tp3=UTSG.integrator(UTSG.DTp3,argsforfunction=[],
            intitial_cond=UTSG.Tp3,time_step=dt)
    UTSG.Tp4=UTSG.integrator(UTSG.DTp4,argsforfunction=[],
            intitial_cond=UTSG.Tp4,time_step=dt)
    UTSG.Tm1=UTSG.integrator(UTSG.DTm1,argsforfunction=[],
            intitial_cond=UTSG.Tm1,time_step=dt)
    UTSG.Tm2=UTSG.integrator(UTSG.DTm2,argsforfunction=[],
            intitial_cond=UTSG.Tm2,time_step=dt)
```

```
    UTSG.Tm3=UTSG.integrator(UTSG.DTm3,argsforfunction=[],
            intitial_cond=UTSG.Tm3,time_step=dt)
    UTSG.Tm4=UTSG.integrator(UTSG.DTm4,argsforfunction=[],
            intitial_cond=UTSG.Tm4,time_step=dt)

    t=t+dt
```

5.3 Data Management

This program will generate state space data of any component at any transient time. So, based on the differential time interval, all the system-governing equations will generate the data points which need to be stored for future investigation for operational and accident dynamics analysis. The number of the data points depends on the variables `simulation_time` and `dt`. Mathematically,

$$\mathrm{N} = \frac{\texttt{simulation time}}{\texttt{dt}} \tag{5.1}$$

For storing these data, a continuous file writing program is necessary as it is a better solution than storing the data in a `list`. It will save a lot of computing memory. For which a program `data_writer.py` is developed. The data system works on a `pandas` and `xlsxwriter` popular python library for tallying the real-time data [22]. The basic structure of the database system is given in Figure 5.2

```
    SUDO code for data_writer program for reactor core
    ---------------------------------------------------
import xlsxwriter
work_book=xlsxwriter.Workbook('test\_case\_reactor\_a.excel')
data_sheet=work_book.add_worksheet("reactor␣name")

row=0
data_sheet.write(row,0,'T')
data_sheet.write(row,1,'Tf1')
data_sheet.write(row,2,'Tf2')
data_sheet.write(row,3,'Tf3')
data_sheet.write(row,4,'Tf4')
data_sheet.write(row,5,'Tf5')
data_sheet.write(row,6,'Tmo1')
data_sheet.write(row,7,'Tmo2')
```

Figure 5.2: Data Management system

```
data_sheet.write(row,8,'Tmo3')
data_sheet.write(row,9,'Tmo4')
data_sheet.write(row,10,'Tmo5')
data_sheet.write(row,11,'Tmo6')
data_sheet.write(row,12,'Tmo7')
data_sheet.write(row,13,'Tmo8')
data_sheet.write(row,14,'Tmo9')
data_sheet.write(row,15,'Tmo10')
data_sheet.write(row,16,'Power')
data_sheet.write(row,17,'Thl')
data_sheet.write(row,18,'Tcl')
data_sheet.write(row,19,'Tlp')
data_sheet.write(row,20,'Tup')
data_sheet.write(row,21,'Precursor')

Simulation_time=20
dt=0.001
t=0
row=1
t=np.linspace(0,Simulation\_time,num=int(Simulation\_time/dt)
   ↪ )
for i in range(len(t)):

    ... ... ... ...
    CALCULATIONS
    ... ... ... ...
    data_sheet.write(row,0,t[i])
    data_sheet.write(row,1,Tf1)
    data_sheet.write(row,2,Tf2)
    data_sheet.write(row,3,Tf3)
    data_sheet.write(row,4,Tf4)
    data_sheet.write(row,5,Tf5)
    data_sheet.write(row,6,Tmo1)
    data_sheet.write(row,7,Tmo2)
    data_sheet.write(row,8,Tmo3)
    data_sheet.write(row,9,Tmo4)
    data_sheet.write(row,10,Tmo5)
    data_sheet.write(row,11,Tmo6)
    data_sheet.write(row,12,Tmo7)
```

```
    data_sheet.write(row,13,Tmo8)
    data_sheet.write(row,14,Tmo9)
    data_sheet.write(row,15,Tmo10)
    data_sheet.write(row,16,reactor.PowerRatio)
    data_sheet.write(row,17,Thl)
    data_sheet.write(row,18,Tcl)
    data_sheet.write(row,19,Tlp)
    data_sheet.write(row,20,Tup)
    data_sheet.write(row,21,reactor.Precursor)
    row=row+1
```

Chapter 6

Results and Case Studies

In this chapter, we present and discuss the results of the study. Initially, we delve into the behaviour of each model or component, including a few case studies for detailed examination. Following this, we explore the interactions in simulations where these components are coupled. The chapter is further divided into two distinct sections, one focusing on the coupled primary circuit components and the other on the secondary circuit components. Lastly, we provide an in-depth discussion on simulations involving all components of the power plant.

6.1 Individual Components

6.1.1 Reactor

Test case 1:

Reactor will be operating at a power of 1000 MWth with a initial Precursor concentration of 3000. The core will have negative moderators and fuel temperature feedback. The hot leg plenum and in the cold leg plenum will have constant temperature. Table 6.1 shows the necessary design data for the reactor core.

Normal Operation No changes in the operational variables. The system will achieve steady state conditions based on the inherent self-control ability due to negative fuel temperature feedback (Doppler feedback), moderator temperature feedback and pressure oscillation feedback.

Table 6.1: Reactor design data

Design Parameter	Symbol	Value
Reactor Diameter	d	3.62712 m
Total core height	h	4.38912 m
active core height	h_active	3.0 m
Fuel diameter	df	0.95e-2 m
fuel pitch	fuel_pitch	1.26e-2 m
Gas gap	G_gap	0.057e-2 m
Fuel density	rho_f	10960 kg/m3
Number of fuel rods	N	9768
Core center factor	Core_f	0.5
RCP position factor	rcp_pos	0.5
power generation factor in fuel	F_r	0.97

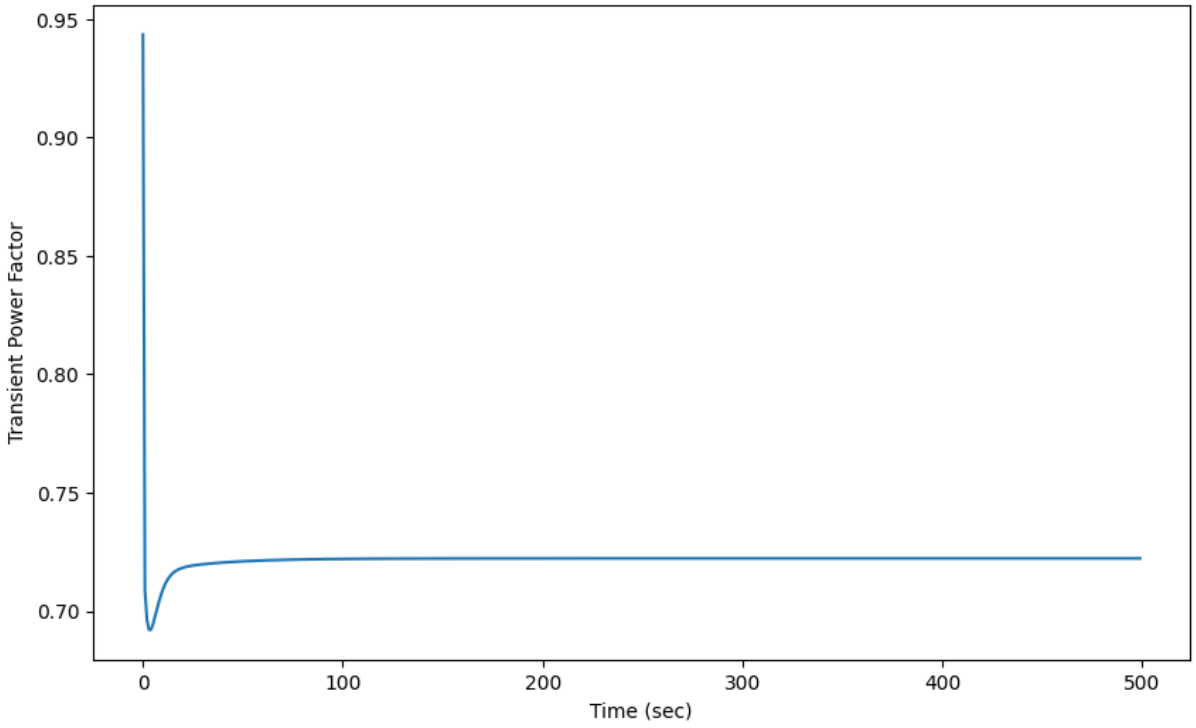

Figure 6.1: Time vs reactor power level

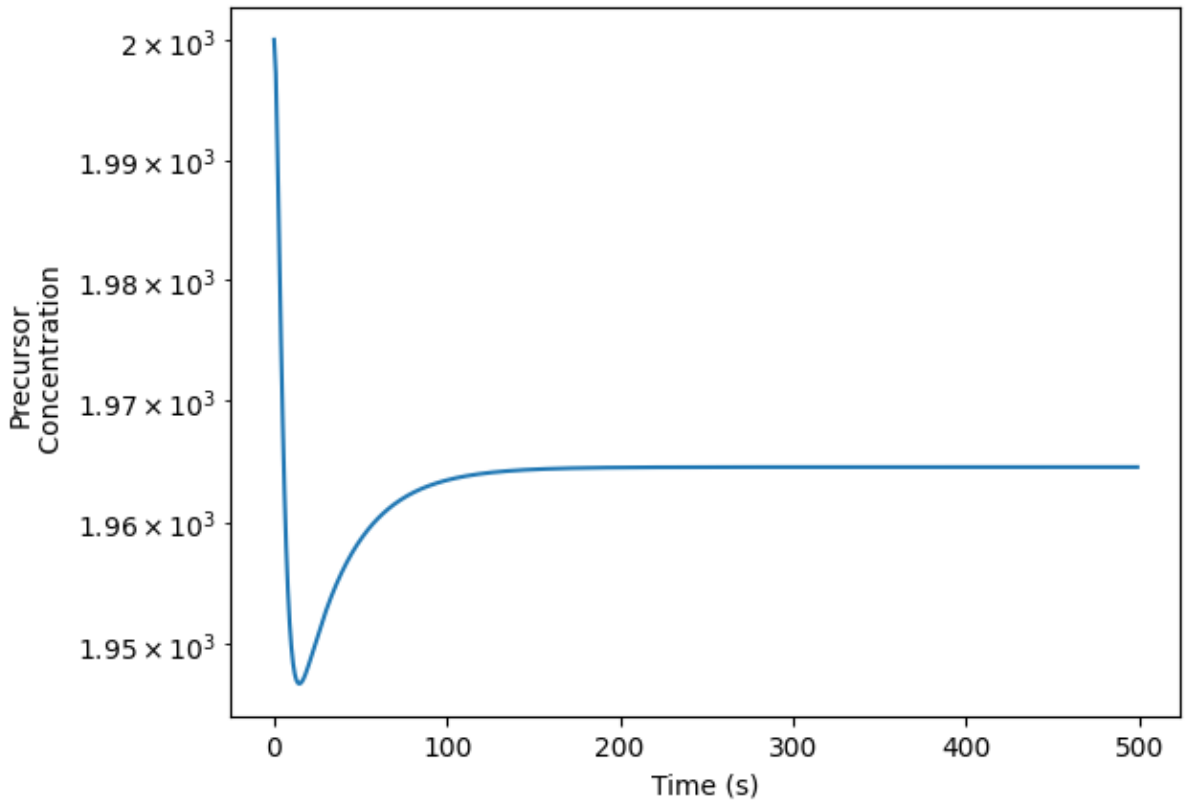

Figure 6.2: Time vs neutron precursor concentration

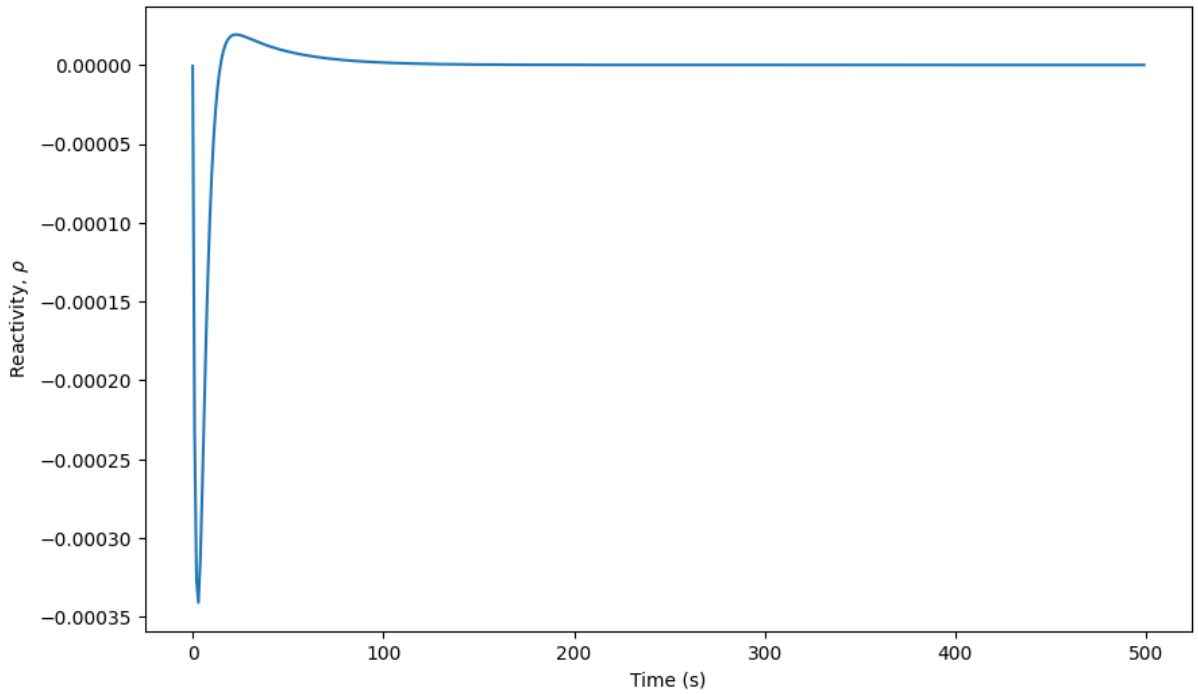

Figure 6.3: Time vs reactivity,ρ

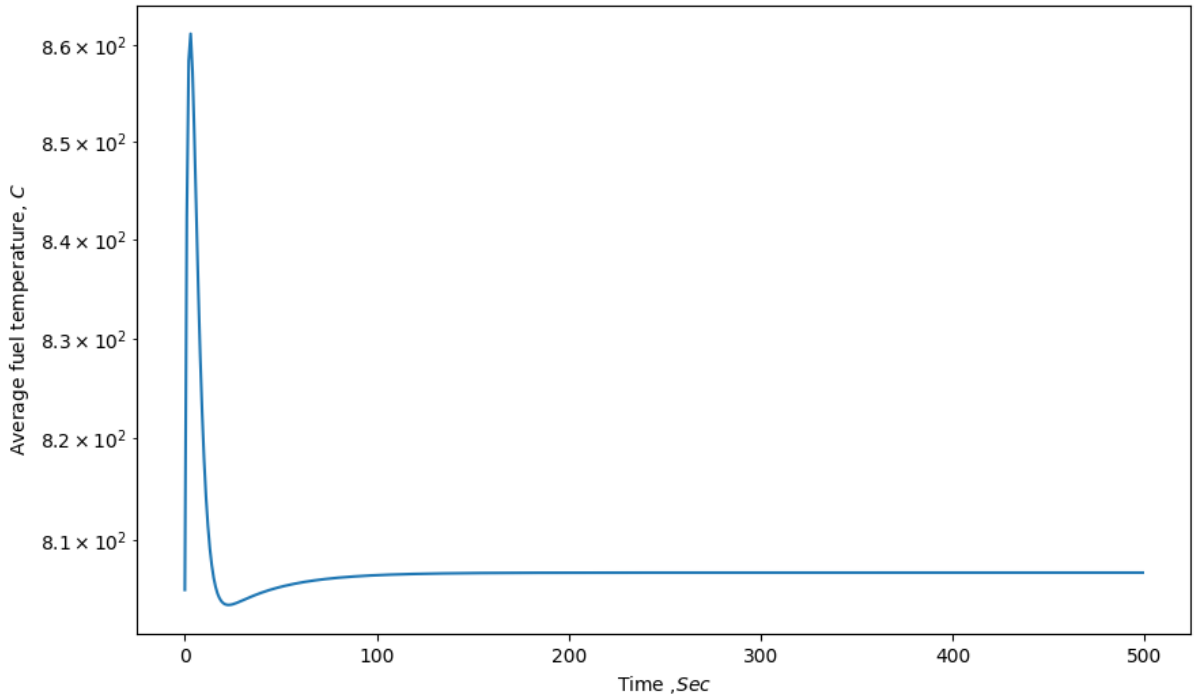

Figure 6.4: Time vs average fuel temperature

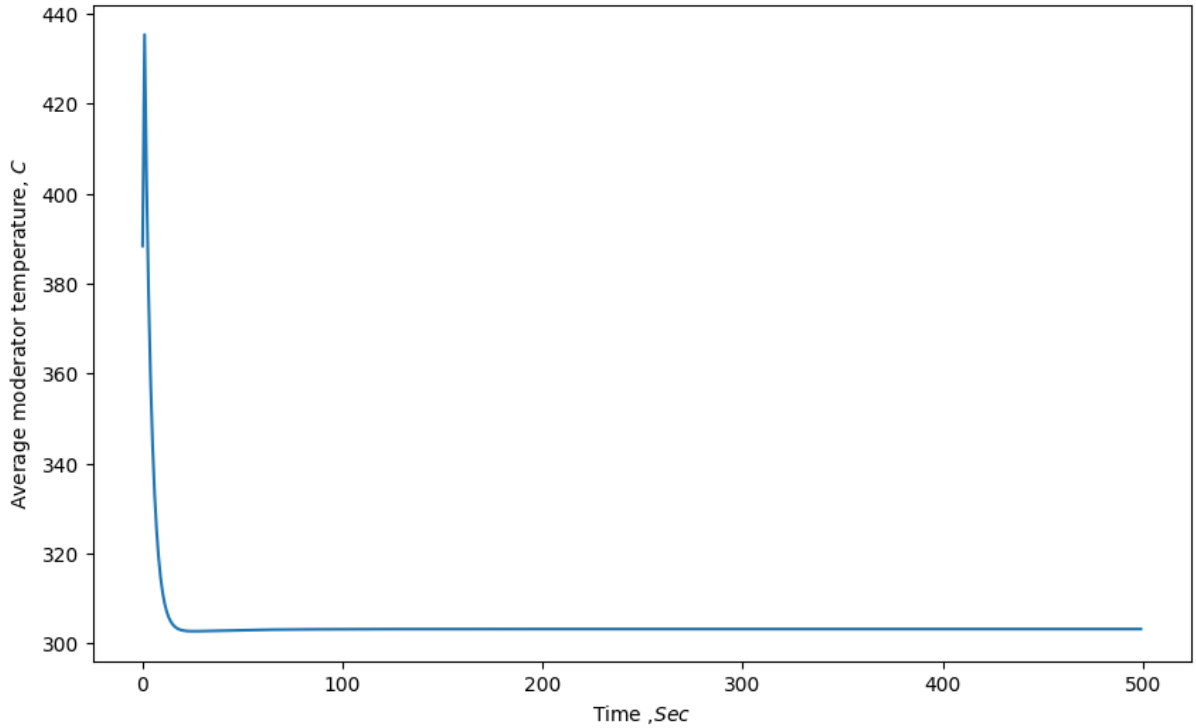

Figure 6.5: Time vs average moderator temperature

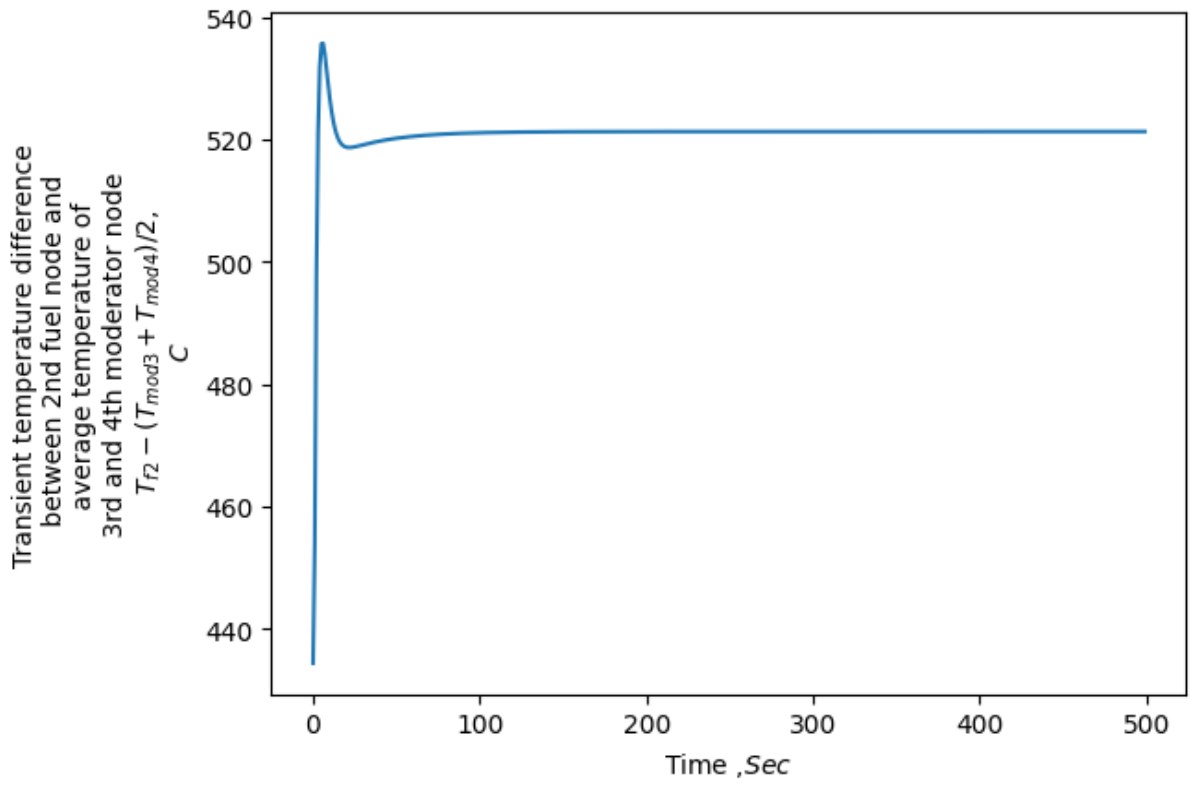

Figure 6.6: Time vs temperature difference in between 2nd fuel node and average temperature of 3rd and 4th moderator node

Test case 2:

The operation will be just like the **Test case 1** but there will be an operation of control rod movement within the reactor core from $[t_1, t_2]$ and $[t_3, t_4]$. Within the time interval of $[t_1, t_2]$ the program `control_rod_driving.py` will set the variable `reactor.ExternalReactivity` to a value of `0.001` and during the time interval of $[t_3, t_4]$ the same program `control_rod_driving.py` will set the variable `reactor.ExternalReactivity` to a value of `-0.001`. Figure 6.7 to Figure 6.12 are the results of case study 2 (`t_1=200,t_2=250,t_3=300,t_4=350`).

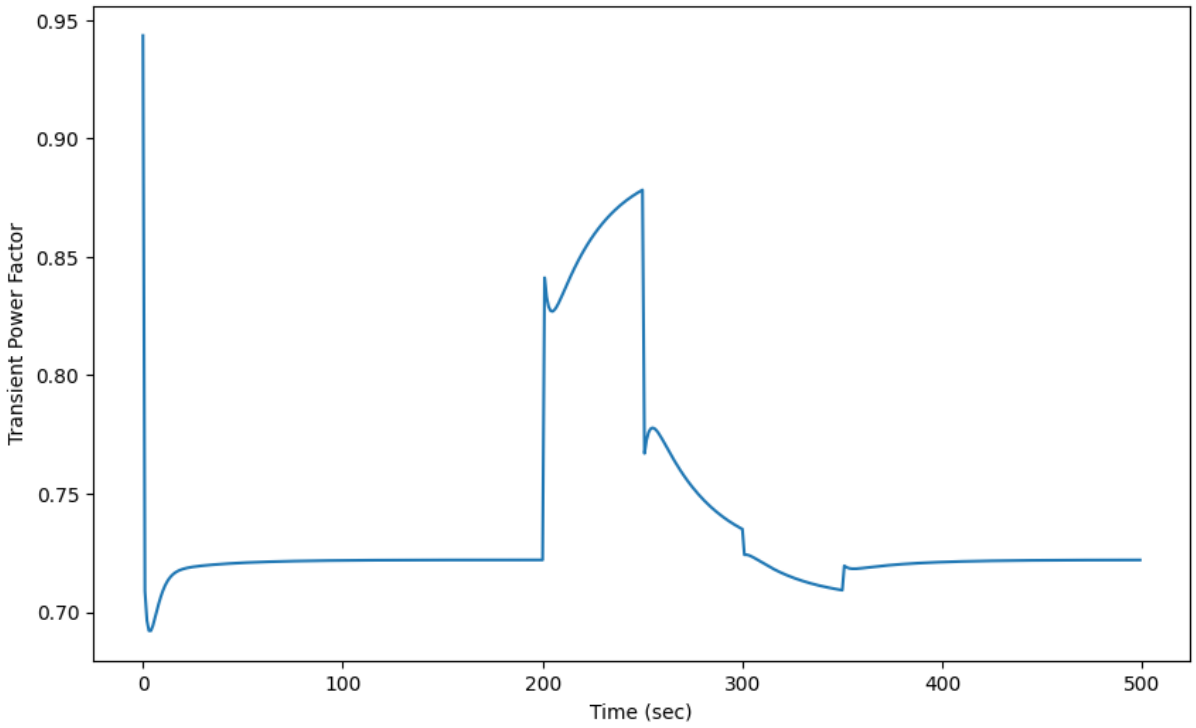

Figure 6.7: Time vs reactor power level

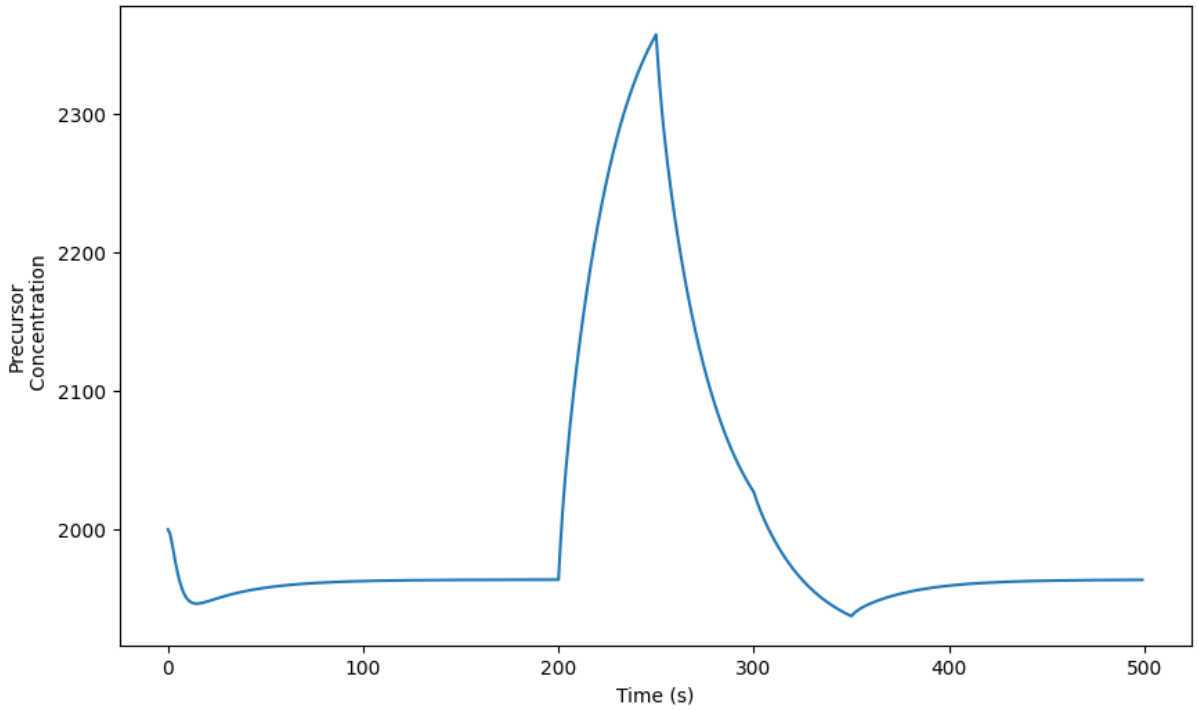

Figure 6.8: Time vs neutron precursor concentration

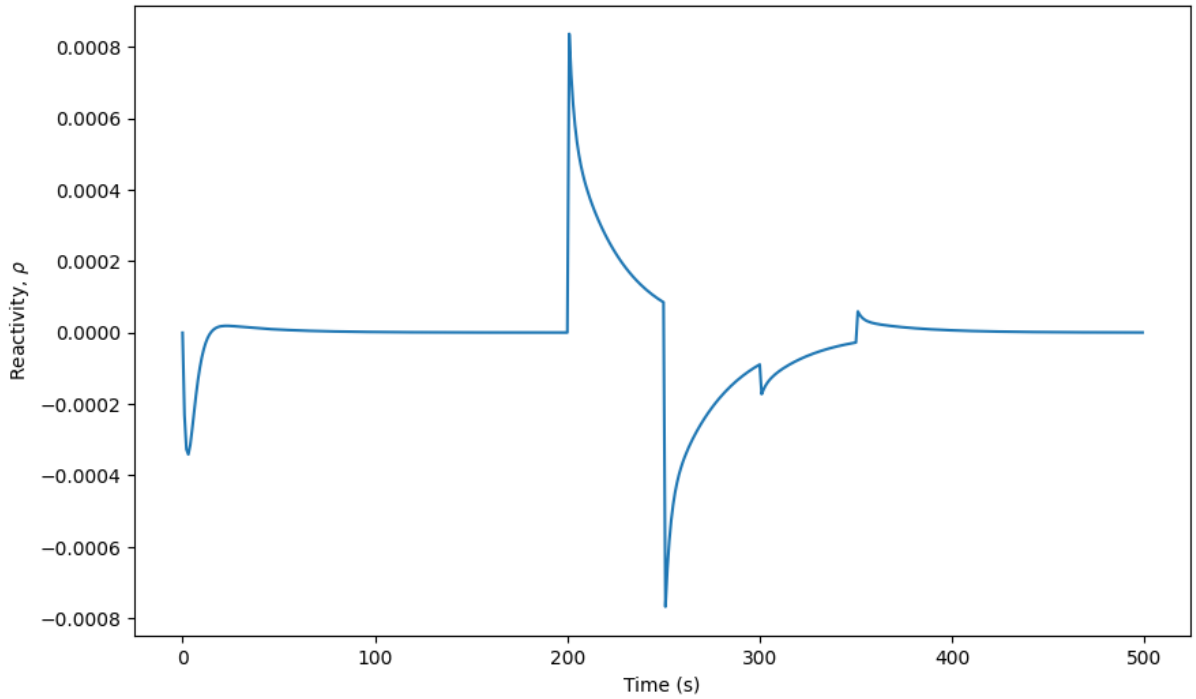

Figure 6.9: Time vs reactivity,ρ

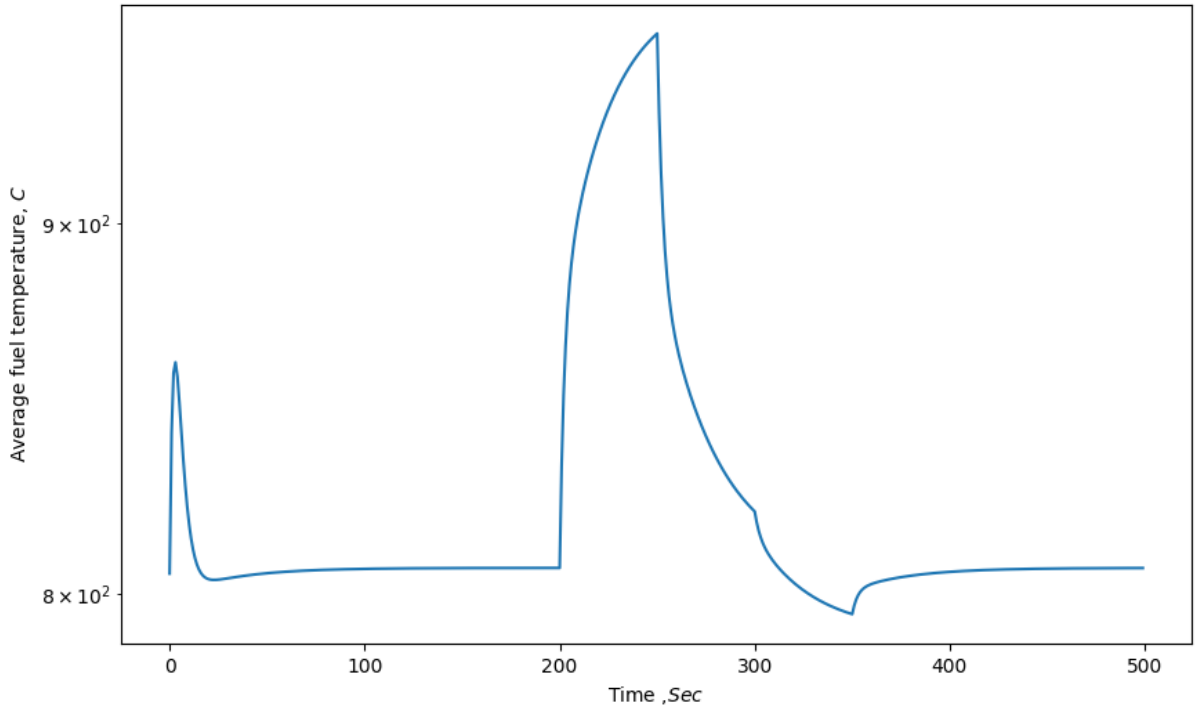

Figure 6.10: Time vs average fuel temperature

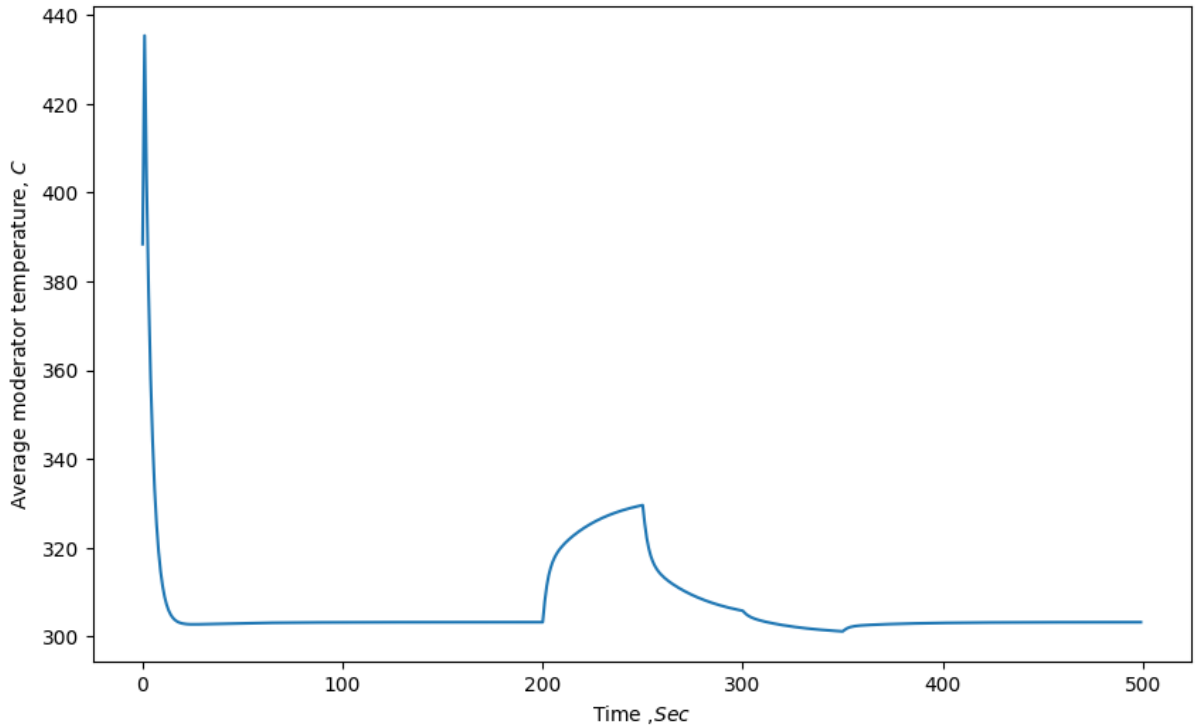

Figure 6.11: Time vs average moderator temperature

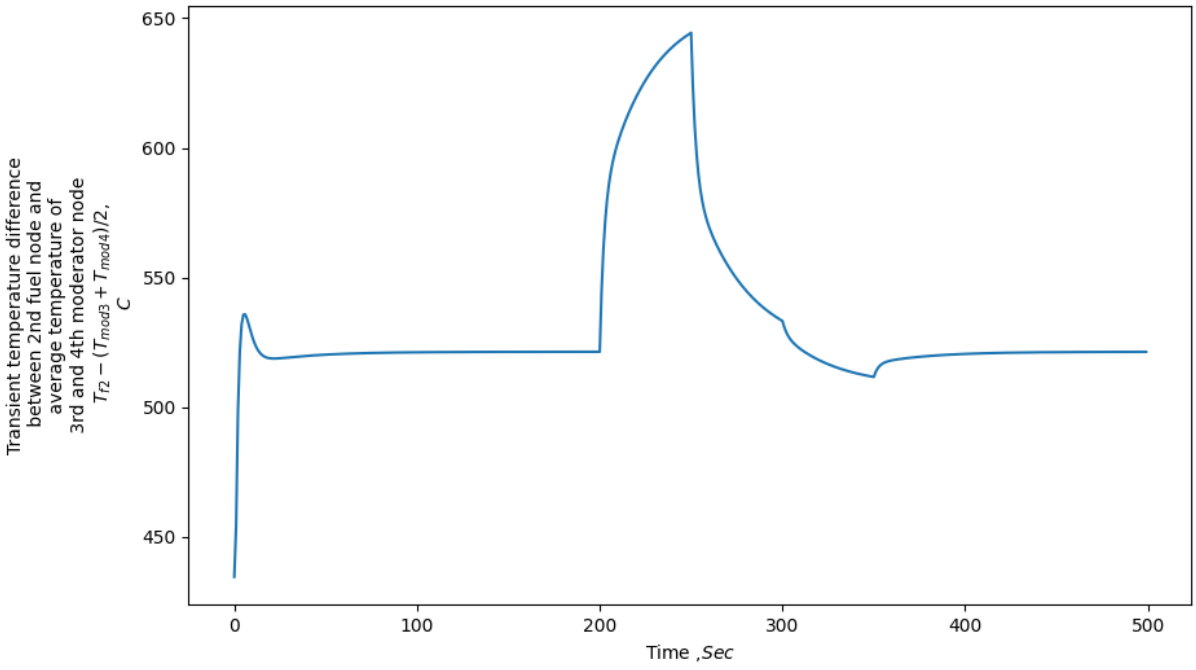

Figure 6.12: Time vs temperature difference in between 2nd fuel node and average temperature of 3rd and 4th moderator node

6.1.2 Pressurizer

The pressurizer system works based on the temperature oscillation feedback inside the reactor core as the surge flow is determined by this formula,

$$\dot{m_{sur}} = \sum_{j=1}^{j=N} V_j \nu_j \frac{dT_j}{dt} \tag{6.1}$$

where $\frac{dT_j}{dt}$ is the time derivative of the transient temperature in the j_{th} moderator or coolant node.
Even though the dynamics of the pressurizer need to be studied in a coupled manner with the reactor core itself for better understating, here are two cases presented that allow us to investigate its behaviour as a single power plant component.

- Test case 1: When the reactor has achieved steady conditions leaving that there are no temperature oscillations with in the reactor core.
- Test case 2: Pressurizer behaviour due to temperature oscillations in the coolant nodes coupled with the reactor core (reactor core test case 2).

Table 6.2: Pressurizer details

Design Parameter	Symbol	Value
Pressurizer diameter	`D`	user defined
Pressurizer height	`l`	14.2524
Sub Cool region height	`l_w`	8.5527
Energy conversion factor	`j_p`	5.4027
Total applied heat	Q	`determined by the pressurizer control algorithm`

Test case 1:

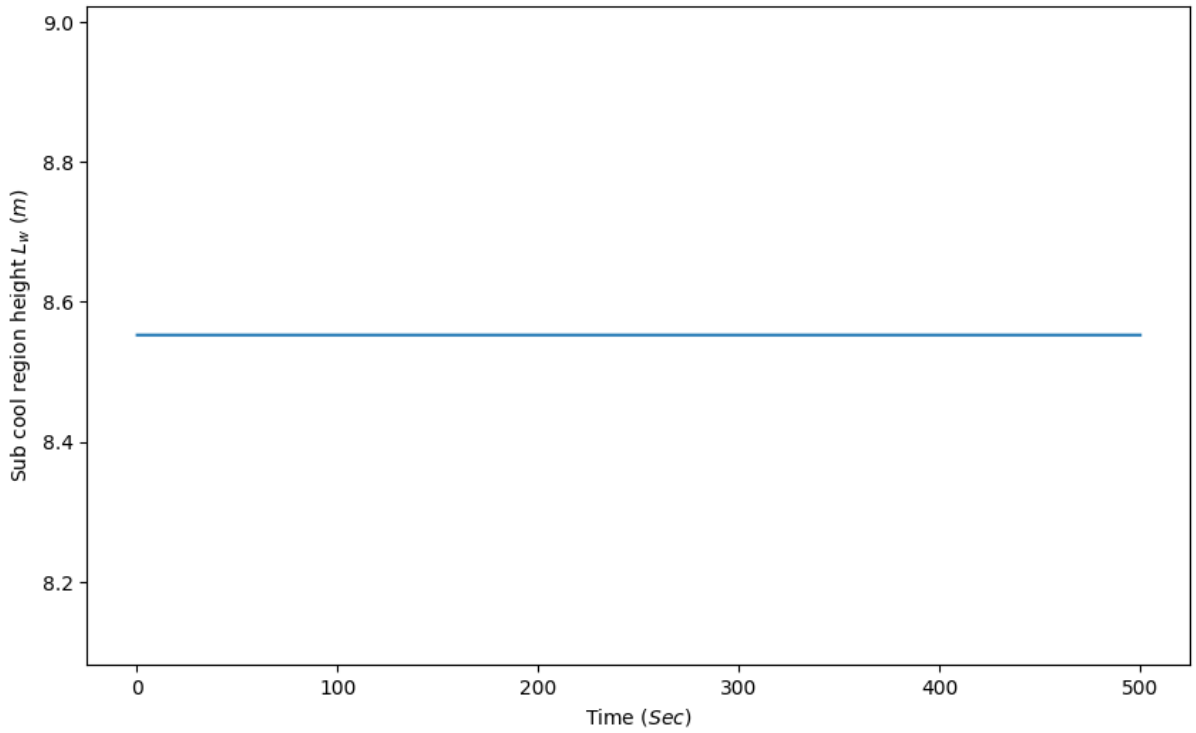

Figure 6.13: Time vs Sub cool region height of the pressurizer

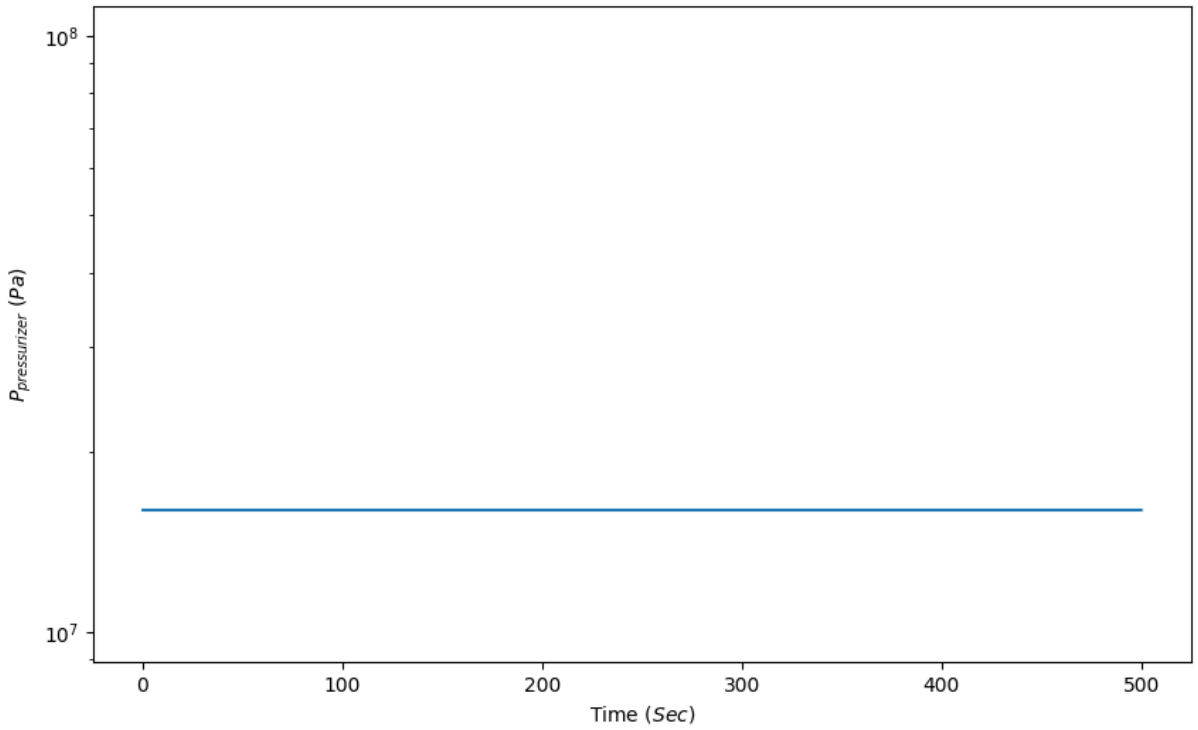

Figure 6.14: Time vs Pressurizer Pressure $P_{pressurizer}$

Test case 2:

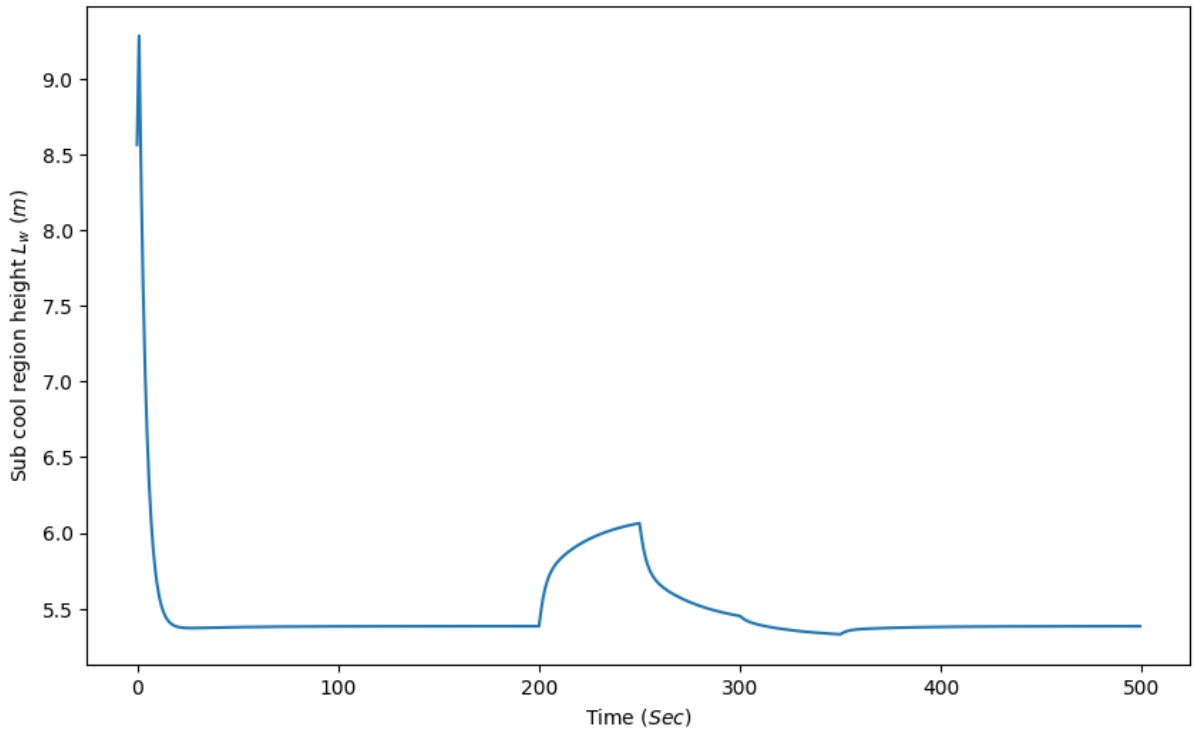

Figure 6.15: Time vs Sub cool region height of the pressurizer

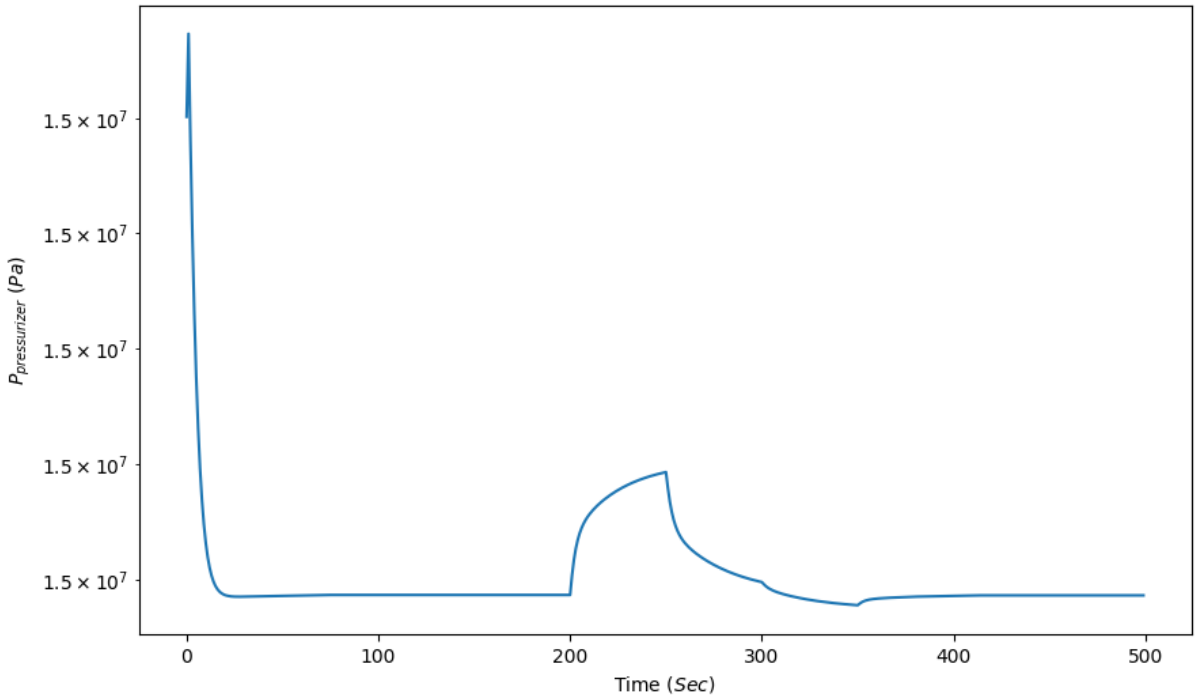

Figure 6.16: Time vs Pressurizer Pressure $P_{pressurizer}$

Table 6.3: Design parameters of UTSG

Design parameters	Variable	Value
Number of U tubes	N	3388
U tube radius (Inner)	R_{in}	0.0098425 m
U tube radius (Outer)	R_{out}	0.0111125 m
Sub cool region length	L_w	3.38
Boiling region length	L_b	7.5
Metal tube conductivity	C_m	460
Primary fluid conductivity	C_{pi}	`Calculated from CoolProps API`
Inlet plenum volume	V_{pi}	Calculated
Outlet plenum volume	V_{po}	Calculated
Inner perimeter of the U tube	P_{r1}	Calculated
Outer perimeter of the U tube	P_{r2}	Calculated
Drum water level	L_w	10.83 m

6.1.3 U tube steam generator

The dynamics of the U tube steam generator is studied in two ways. The first case study is done under steady state condition where the hot leg temperature of the reactor core is presumed to constant (The reactor will be in equilibrium) and the 2nd case study is coupled with the reactor core where the reactor will operate under constant pressure conditions (No pressure oscillations with in moderator nodes of the reactor core). The important parameters and the initial conditions of the UTSG are given in the following table 6.3 and respective test case.

Test case :

USTG operation considering steady conditions in the other primary loop components.

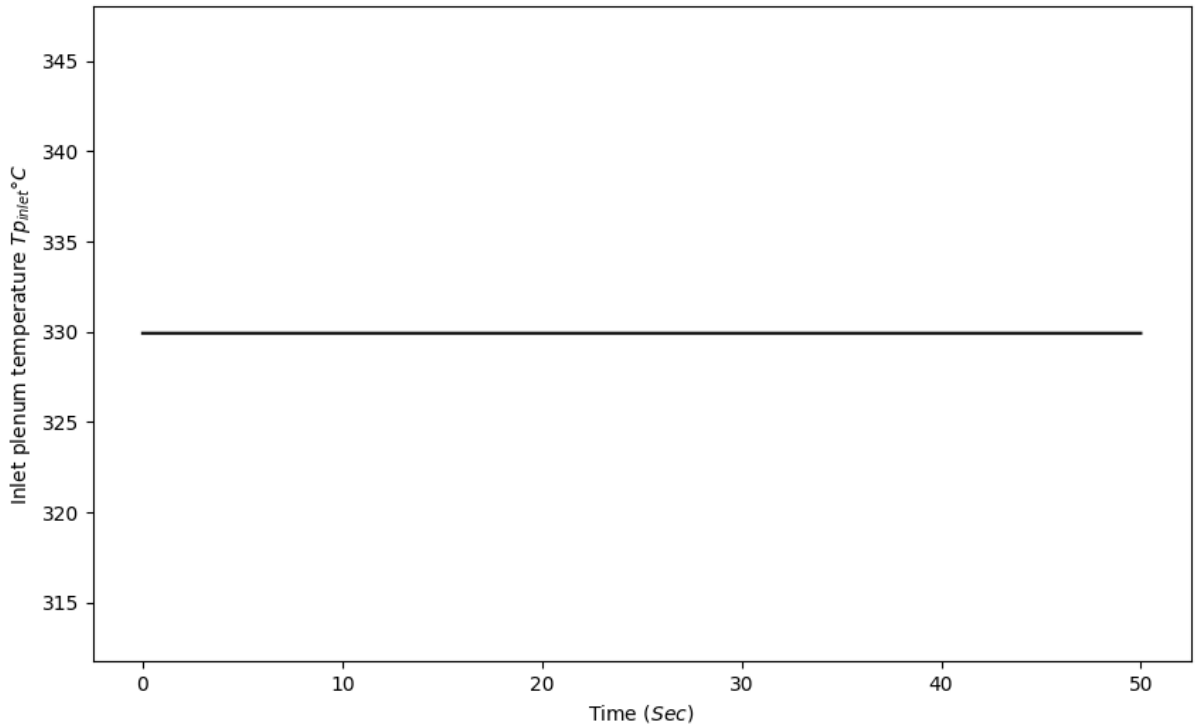

Figure 6.17: Time vs Inlet plenum temperature

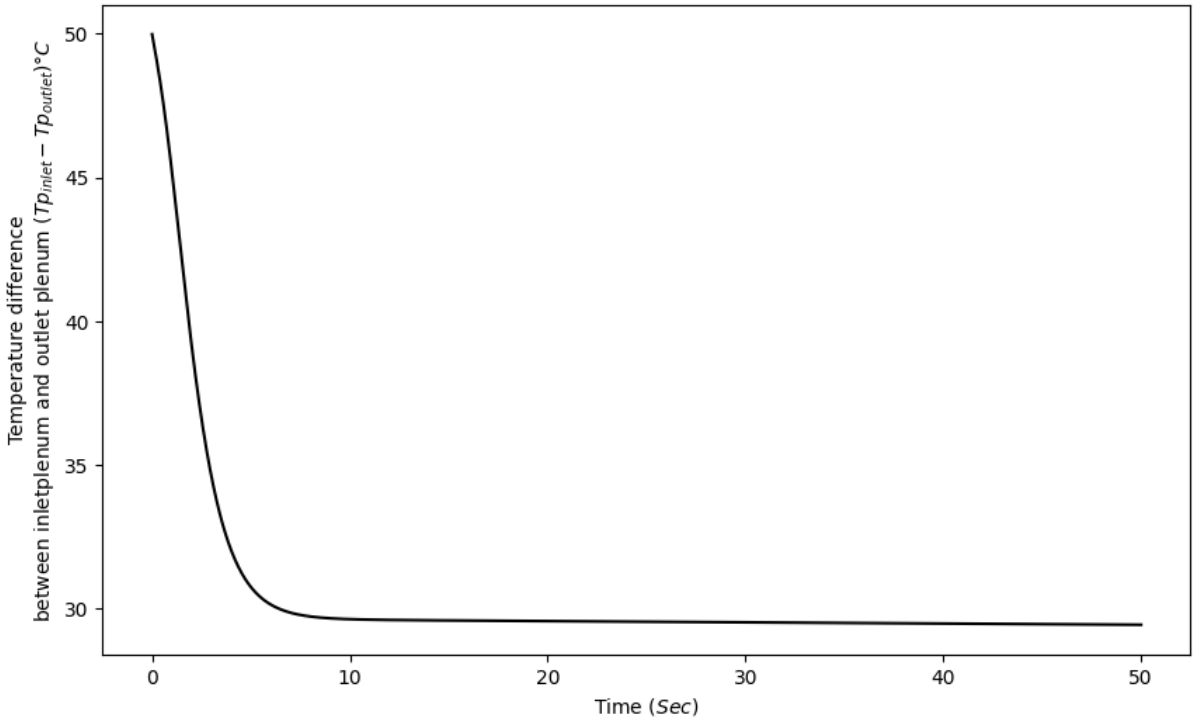

Figure 6.18: Time vs Temperature difference in inlet and outlet temperature of the UTSG

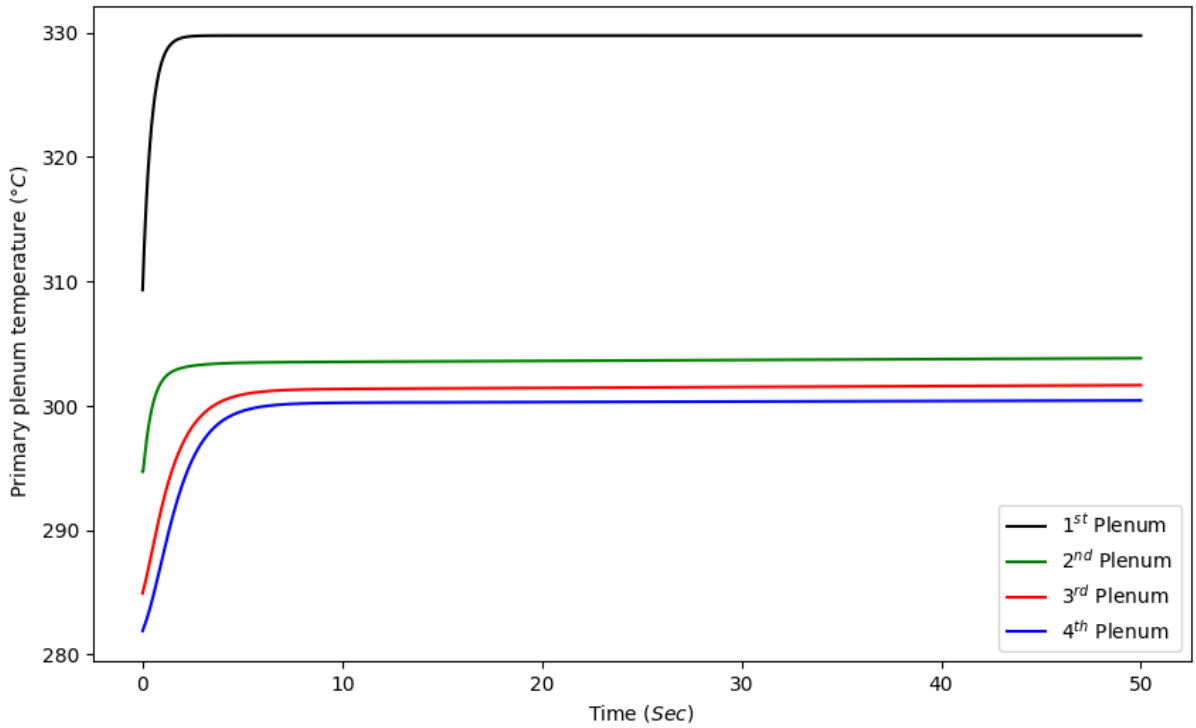

Figure 6.19: Time vs Primary Plenum temperatures

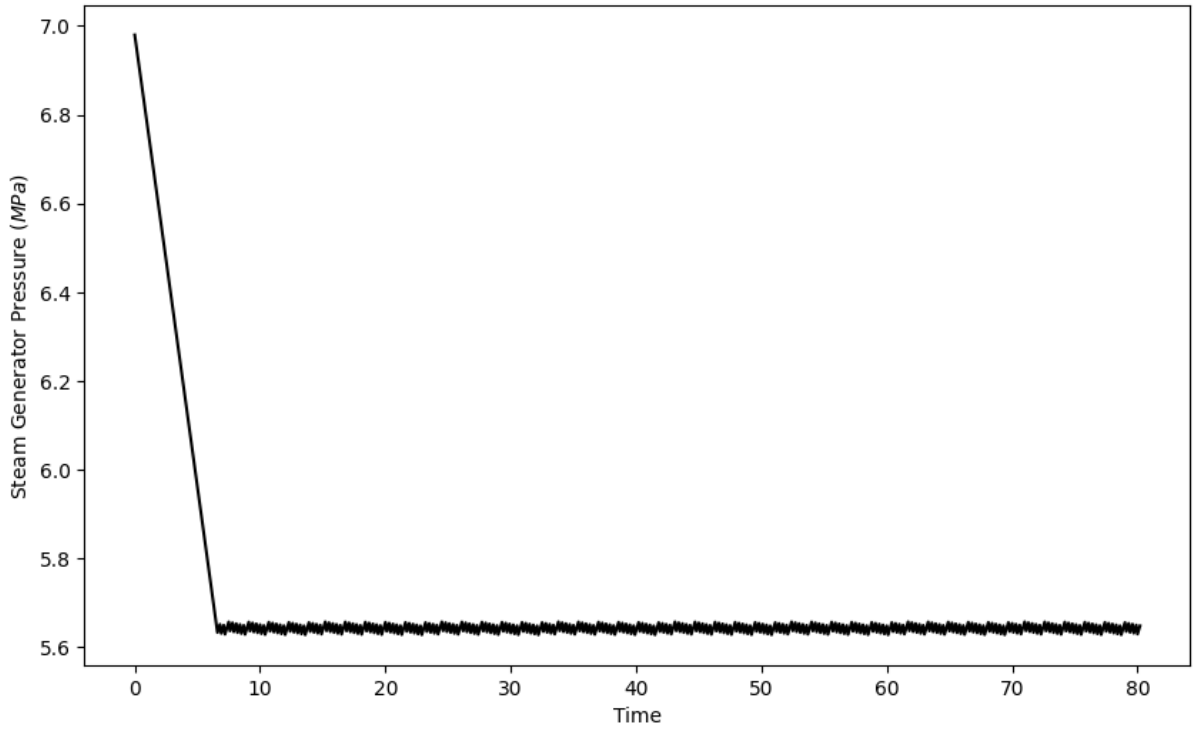

Figure 6.20: Time vs steam generator pressure

6.2 Coupled modeling:

6.2.1 Test case 1:

Reactor startup test with coupled pressurizer and steam generator model. The reactor operation will involve no control rod. The system will achieve equilibrium based on the embedded control system with in the reactor core.

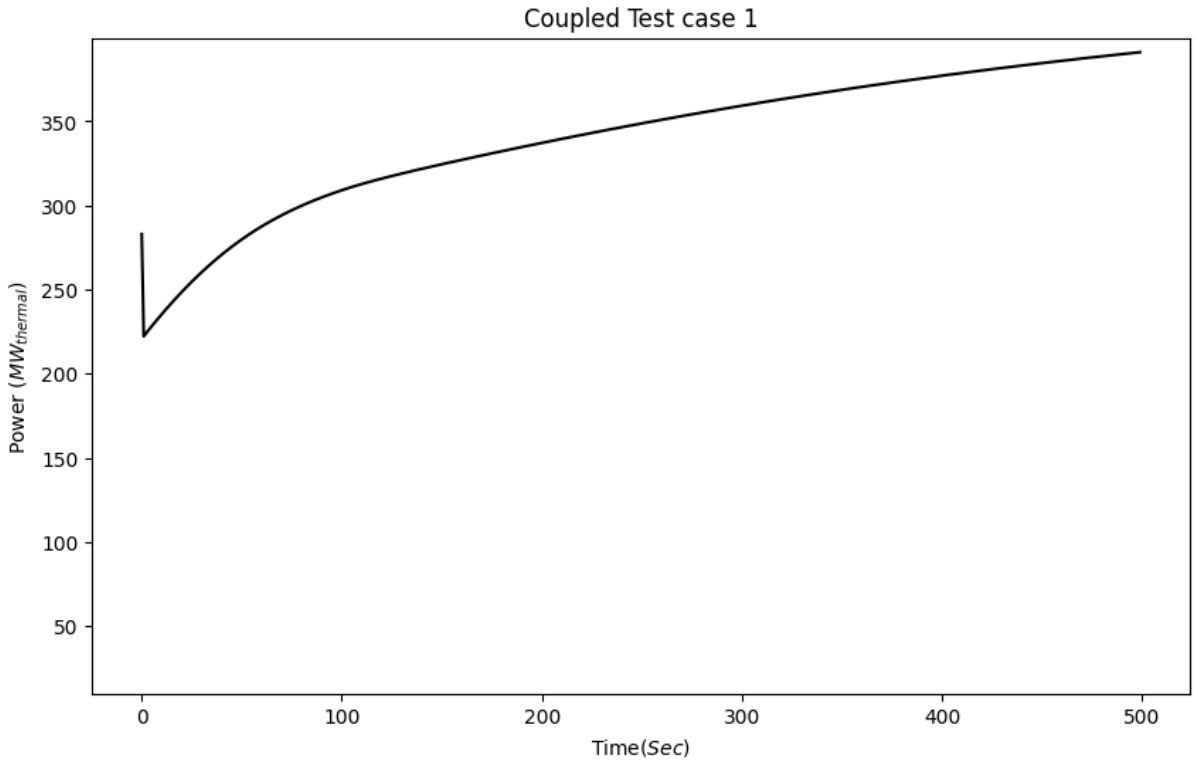

Figure 6.21: Time vs reactor power

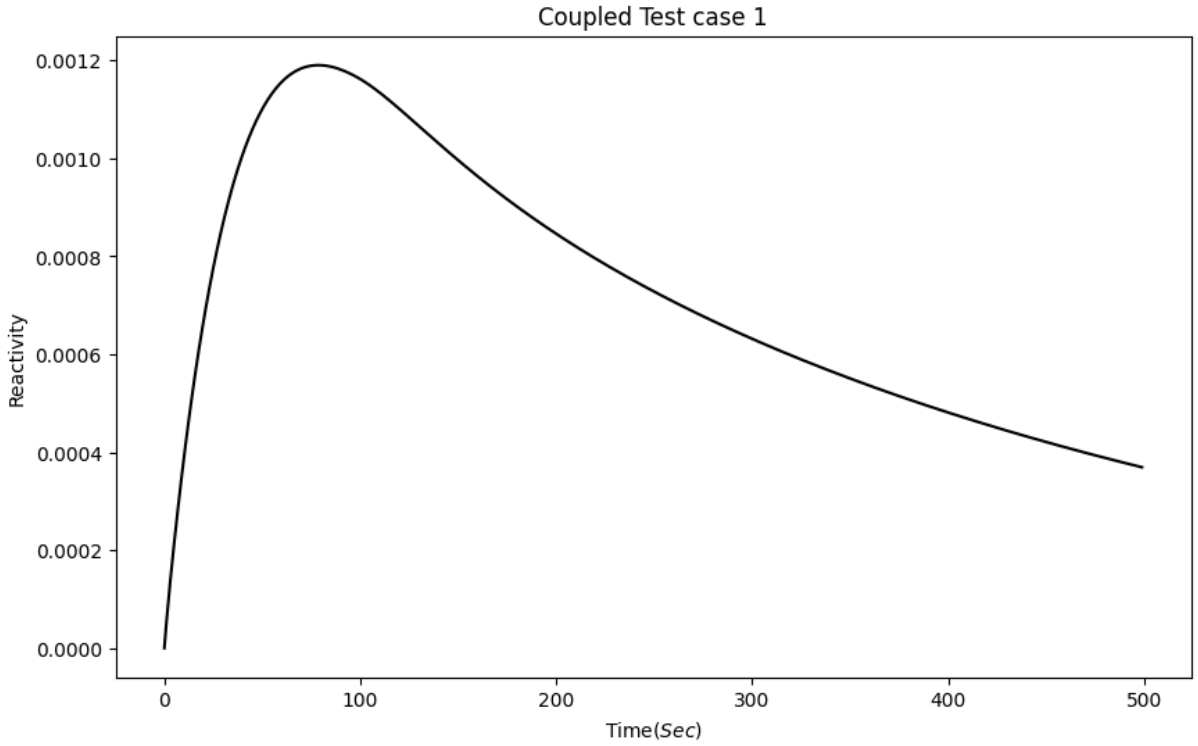

Figure 6.22: Time vs reactivity ,ρ

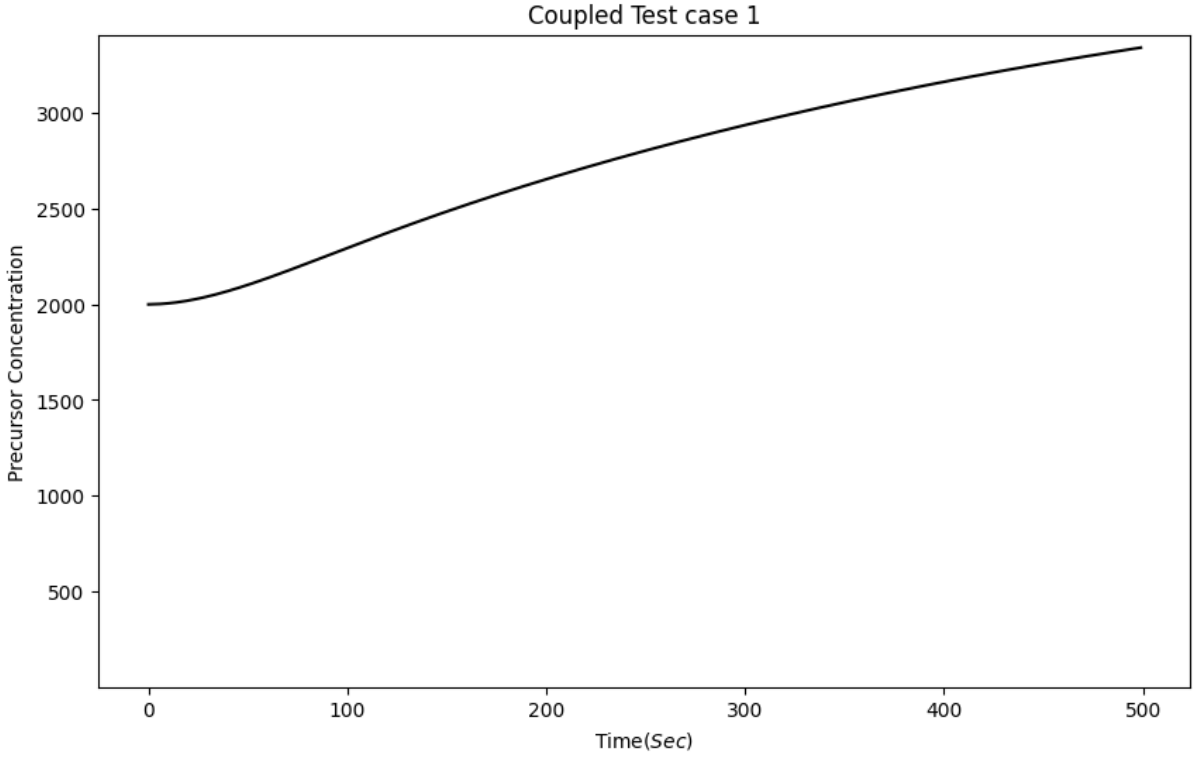

Figure 6.23: Time vs Precursor concentration

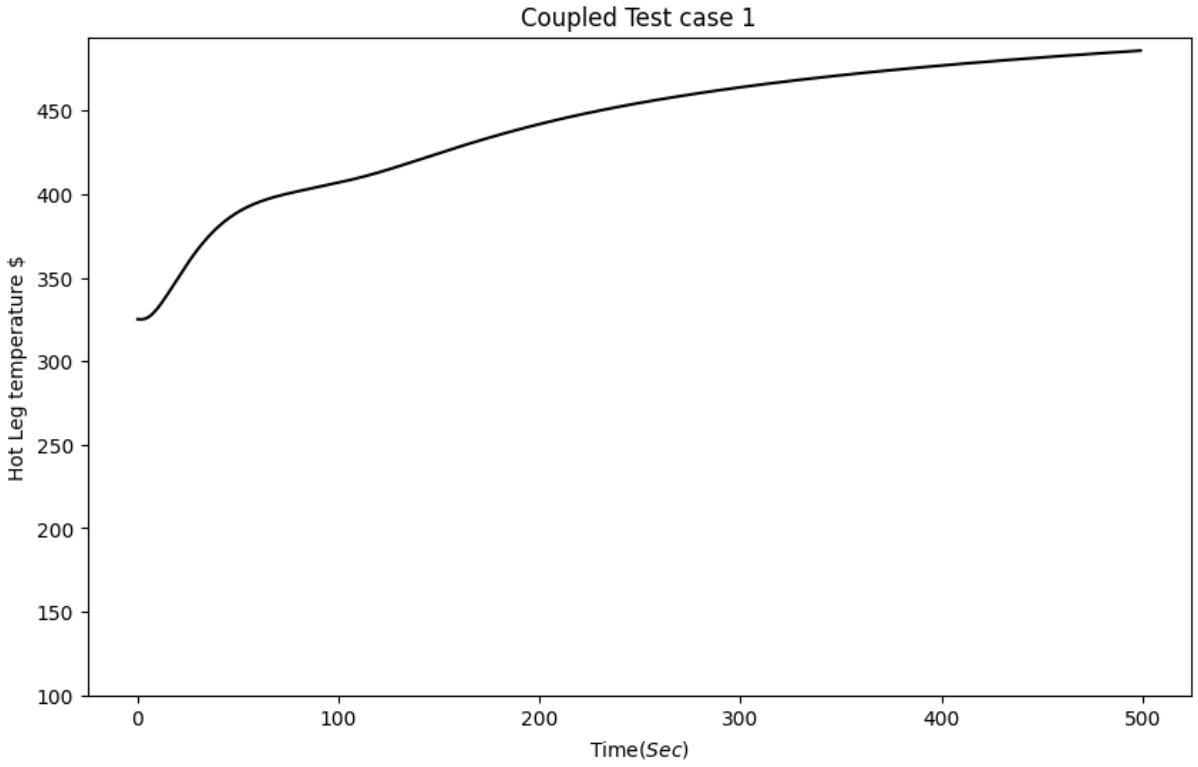

Figure 6.24: Transient evolution of hot leg temperature

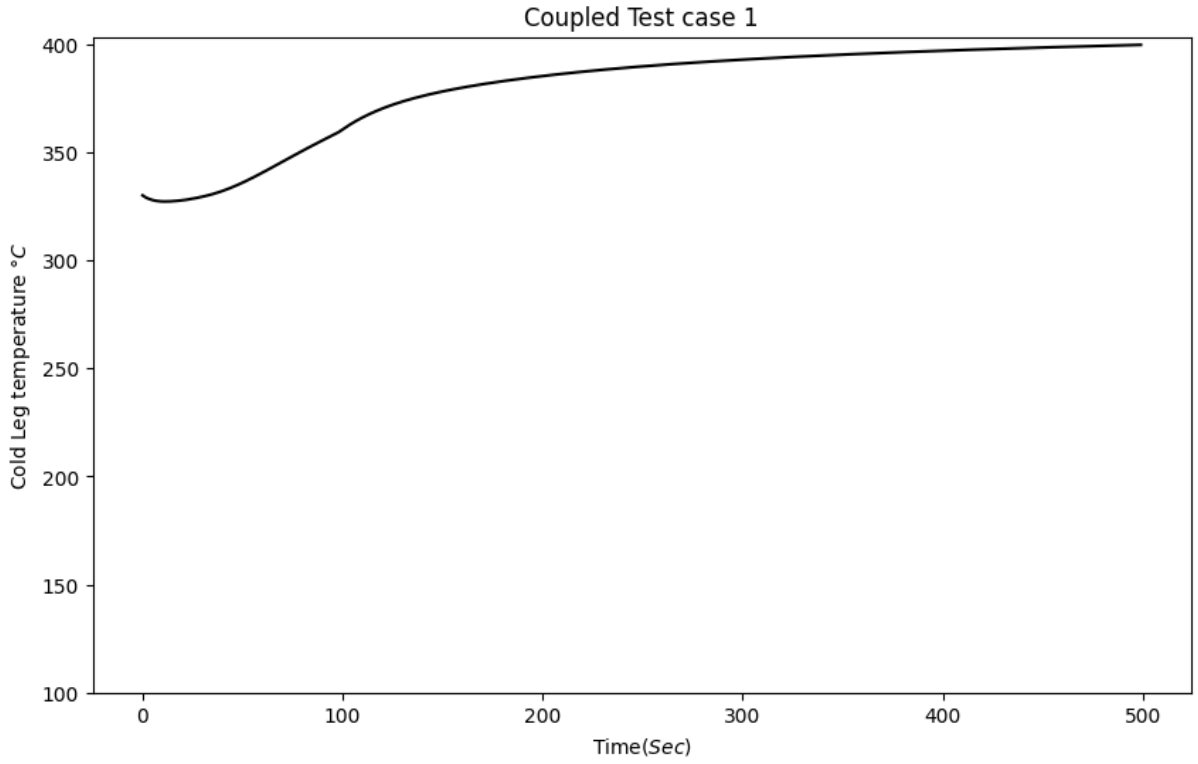

Figure 6.25: Transient evolution of cold leg temperature

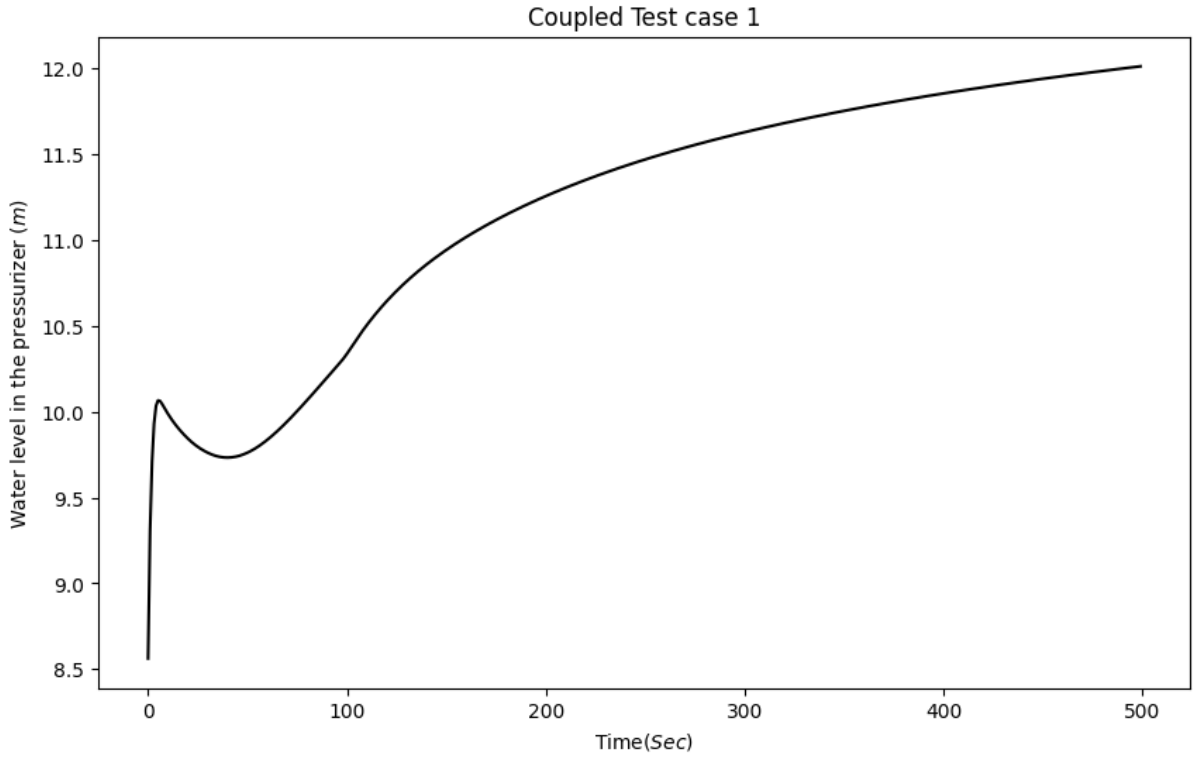

Figure 6.26: Transient behaviour of water level in the pressurizer

**No spray or heater used to control the pressure level as the goal is to investigate how pressure Oscillations happened in the rector core*

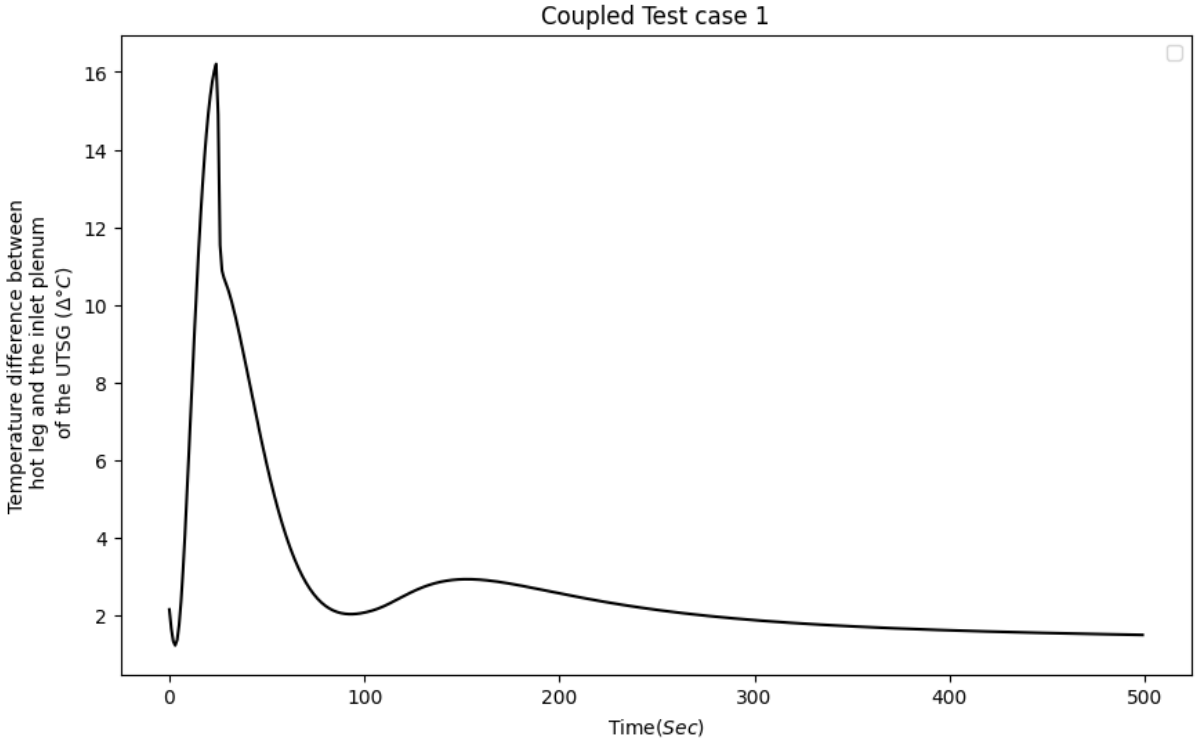

Figure 6.27: Transient behaviour reactor hot leg and UTSG inlet temperature difference

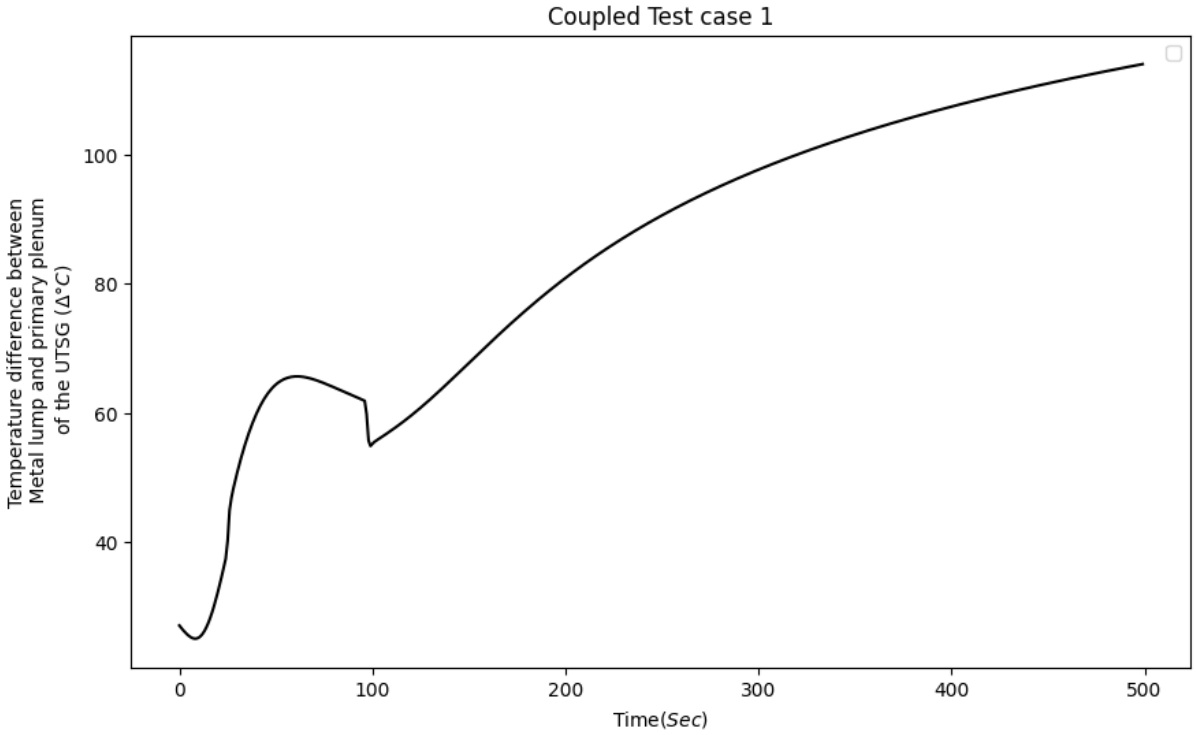

Figure 6.28: Transient behaviour UTSG primary plenum and metal plenum temperature difference

6.2.2 Test Case 2

The operation of this coupled model with UTSG, pressurizer and rector core will be the same as the test case 1 but here during$[t_2 = 200sec, t_3 = 300sec)$ time period there will be an control rod withdrawing operation and from $[t_3 = 300sec, t_4 = 400sec)$ there will be an control rod inserting operation.

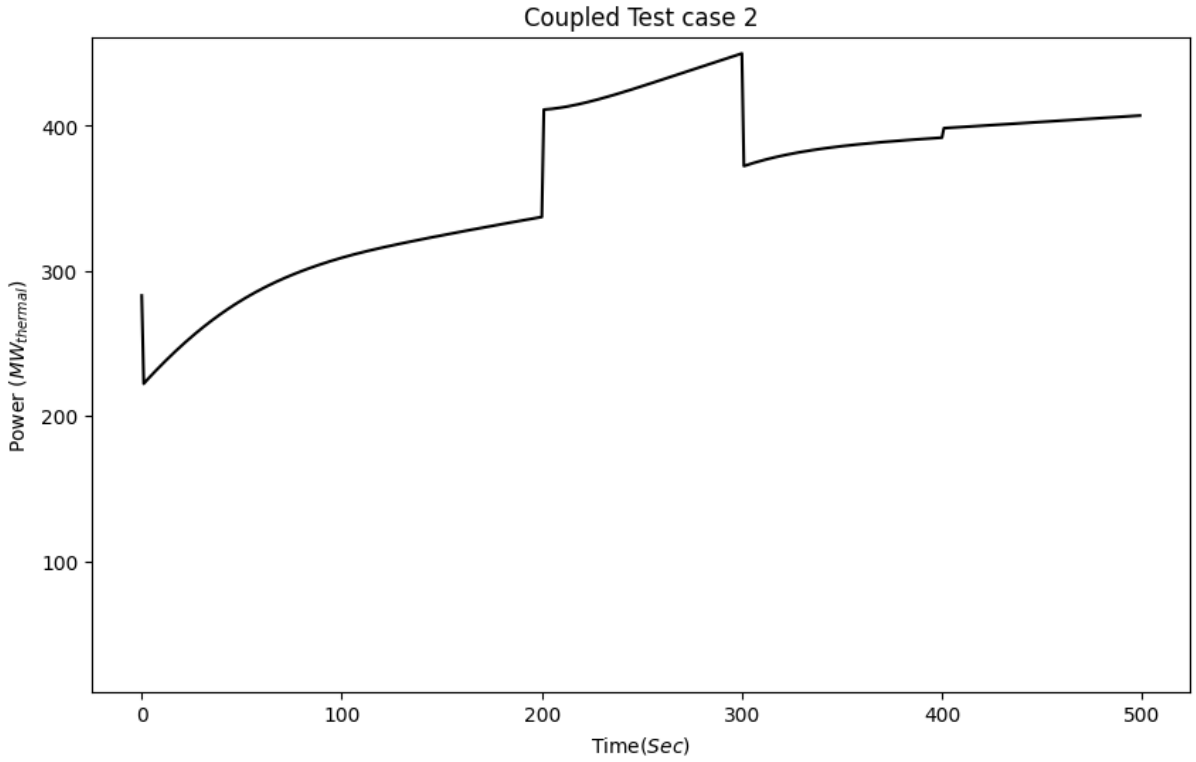

Figure 6.29: Transient behaviour reactor power

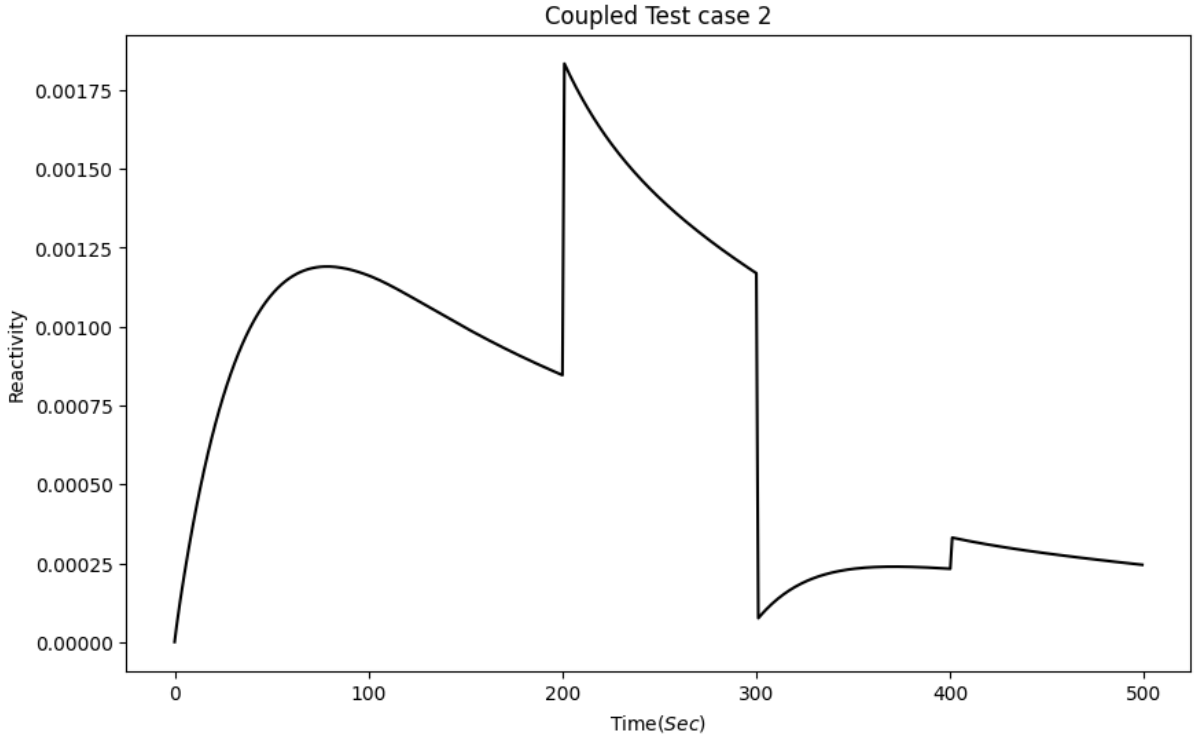

Figure 6.30: Transient behaviour of reactivity ,ρ

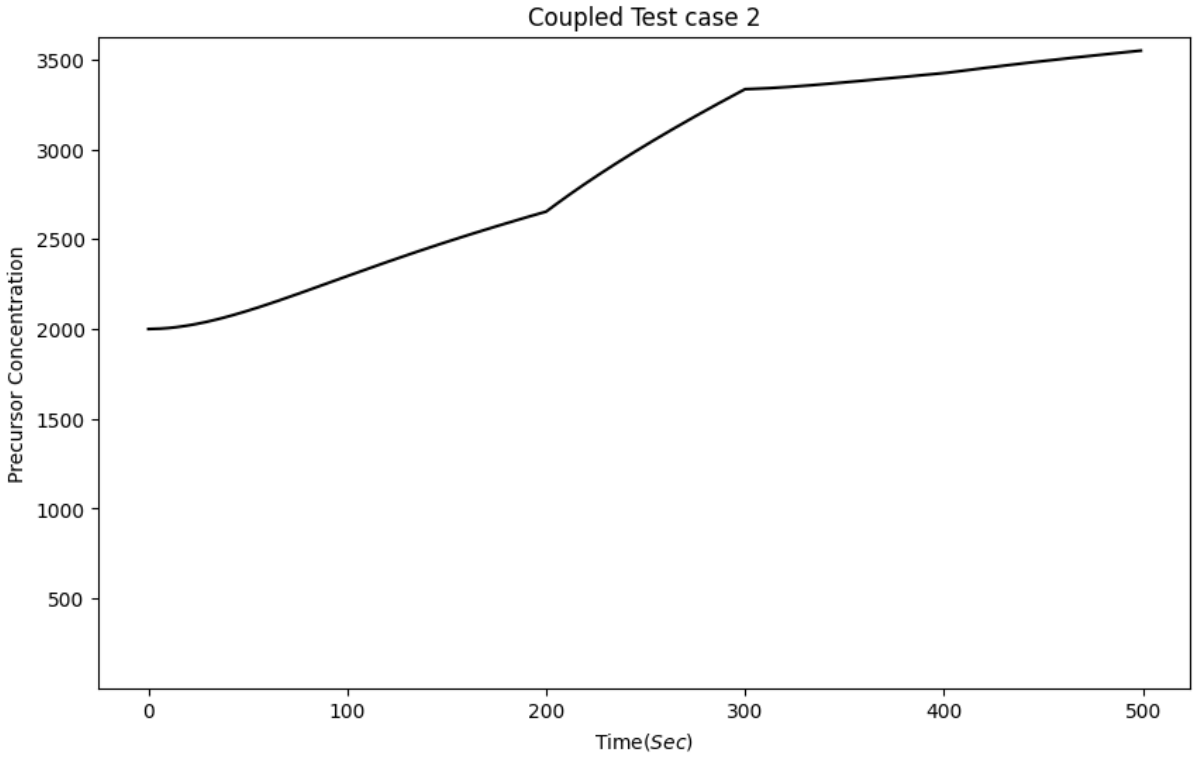

Figure 6.31: Transient behaviour of Precursor concentration

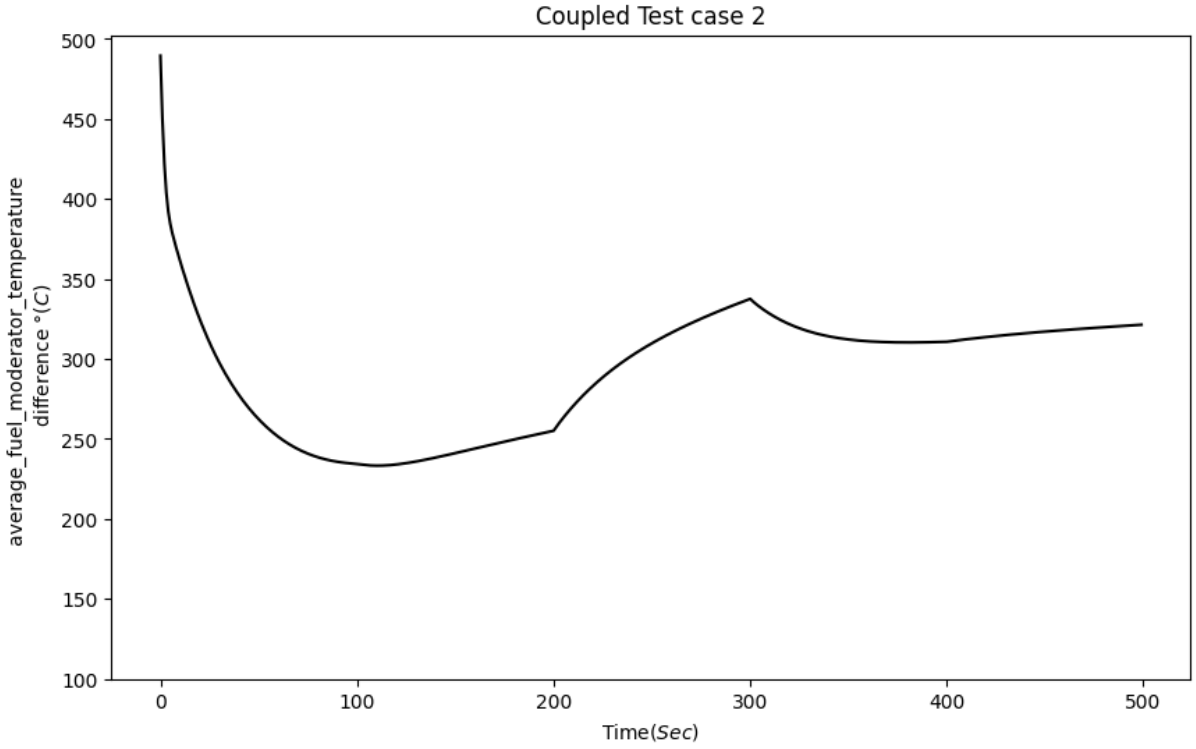

Figure 6.32: Time vs average fuel and moderator temperature difference

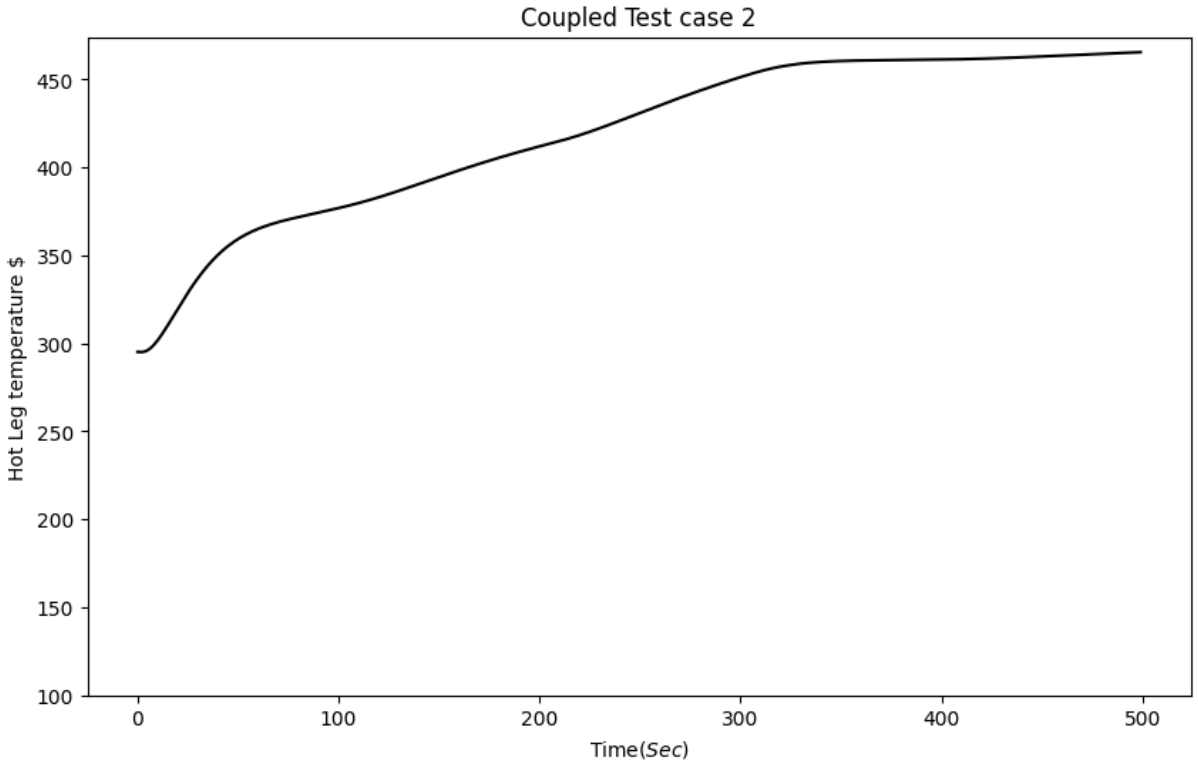

Figure 6.33: Transient evolution of hot leg temperature

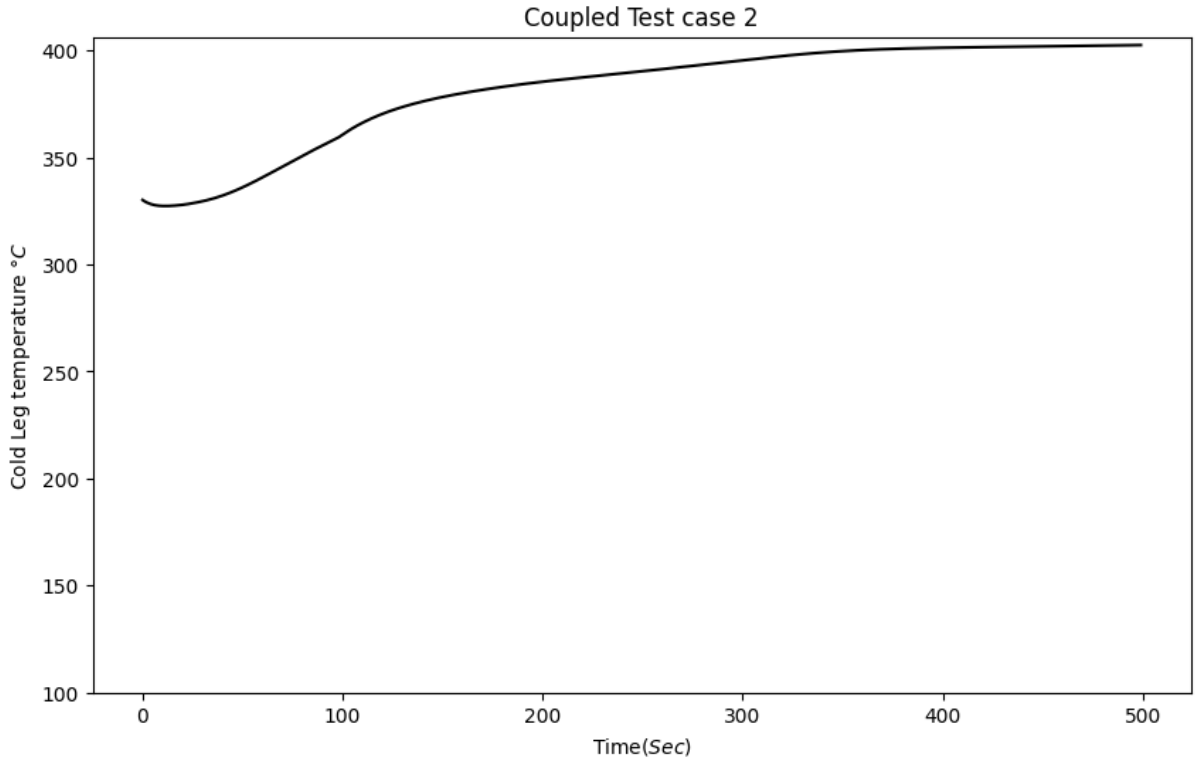

Figure 6.34: Transient evolution of cold leg temperature

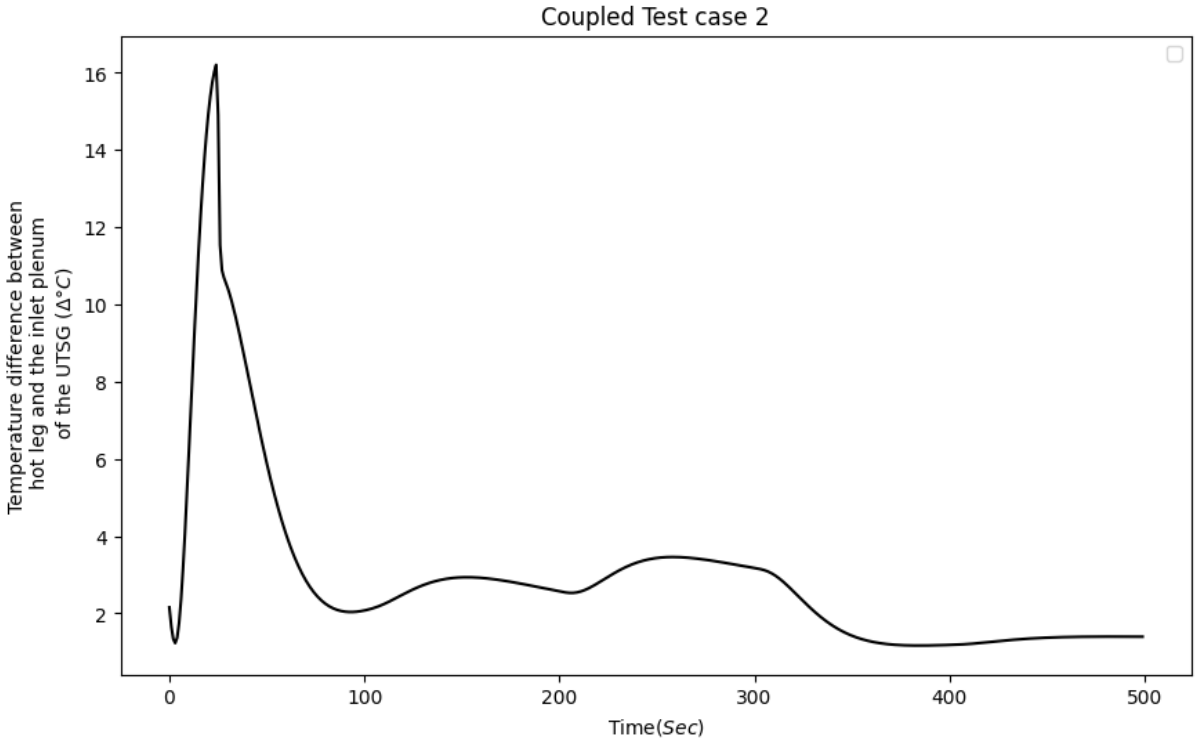

Figure 6.35: Time vs hot leg and UTSG inlet plenum temperature difference

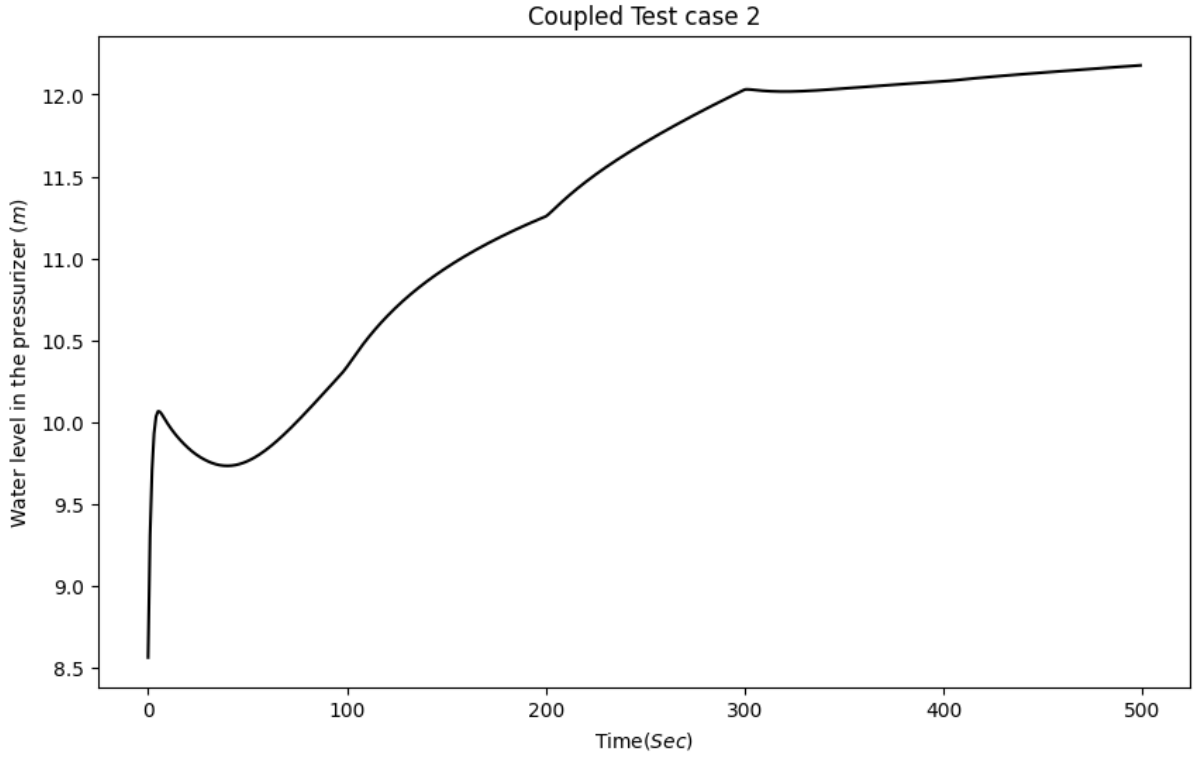

Figure 6.36: Transient stabilization of pressurizer water level

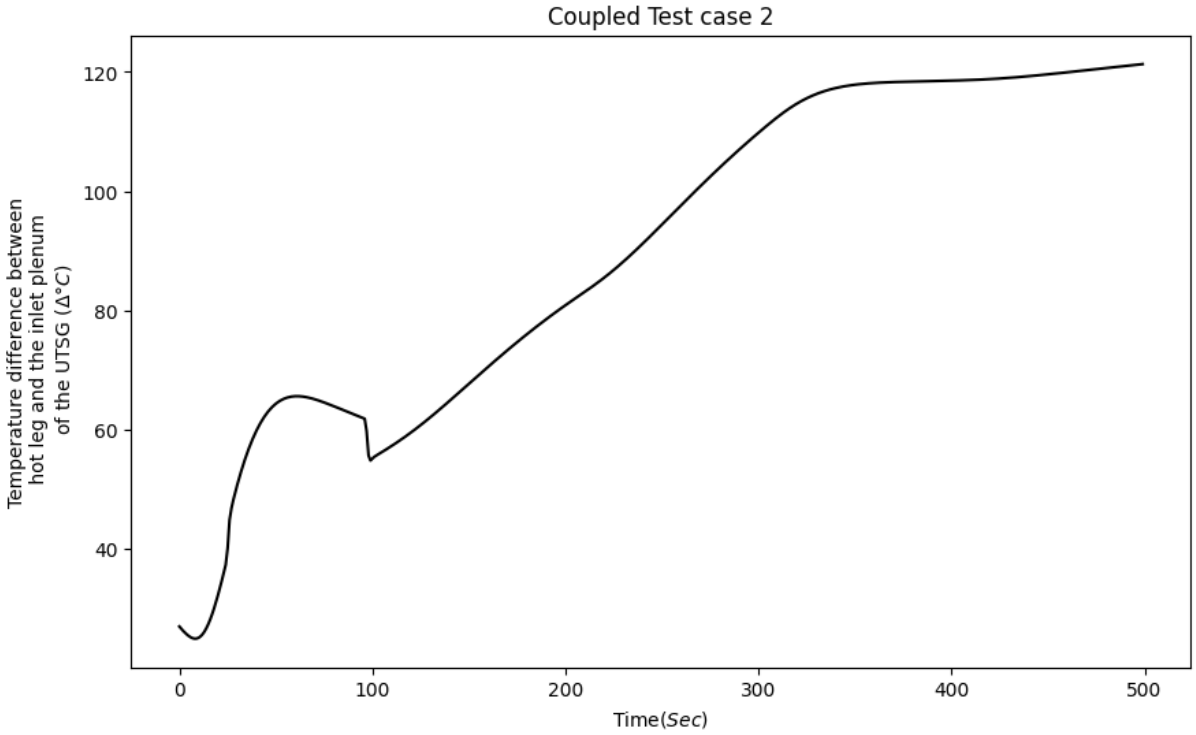

Figure 6.37: Transient behaviour UTSG primary plenum and metal plenum temperature difference

Chapter 7

Conclusion

PWR technology has been in the world for more than 60 years and it's very mature now. However, three major nuclear accidents over the last 40 years have fueled the details investigation of nuclear power safety. The key lies in the proper operation of PWR reactors maintaining the safety limits for which understanding the reactor system is necessary. The main this study is to develop a program which offers more flexibility to simulate any kind of operational situation.

7.1 Results of this study

In this research, a nonlinear dynamic model has been meticulously developed to describe the behaviour of a PWR system. The investigation initially focuses on assessing the individual and couple components of the model, which include the reactor core, steam generator, and pressurizer. Individual models are studied assuming that other components are in equilibrium. The initial conditions for each state space variable of each components in each simulation are the same for individual and coupled modeling.Randomness is applied to the input parameters to evaluate the response of each component. By comparing the simulation results with data from previous studies [5], [4],[6], it is evident that these component models exhibit realism and have the capability to accurately predict dynamic responses. Once the validation of these individual component models is accomplished, the study proceeds to combine them with the hot leg riser and down comer to form a comprehensive model for a single PWR unit. Subsequently, this complete PWR unit model is put to the test by subjecting it to independent step changes in input variables. It's important to note that, since this research is original, there is no existing data available for direct comparison. However, the results obtained from these tests are found to be in good agreement with theoretical expectations, affirming the model's reliability. For example the moderator temperature inside the reactor core is about 300 degree C and average fuel temperature for $1000MW_{th}$ reactor is 800 degree C [23].

There were no control algorithms. The reactor model studied in this thesis, achieved equilibrium conditions due to the self-controllability embedded with the PWR core as a feature of negative fuel, moderator temperature and positive pressure feedback co-efficient.

7.2 Future works

The program developed in that study is still in beta phase. Even after this dissertation, it will need some major updates and the addition of new secondary loop components. Developing a generalized PWR operation program should be collaborative given that it needs a lot of design parameters and they depend on the design. Component-specific control programs are also necessary to get a more realistic simulation of the power plant. Possibly PID controller program with feedback control loop system. But there are a few things which can be done to make this program run more smoothly and accurately on any computer or workstation.

- **Multiprocessing and multi-threading.** (As this program solves the non-linear differential equations on the fly running the simulations on a single processor is computationally more expensive. If the parallelization of this program can be done, the different components will run on different threads/processors and an intermediate database system to track and give the transient feed of one component to other components. This is how the program will run more smoothly.)
- **User interface development** (Currently the program is based on the AZOG API system. A simulation-like UI will improve the flexibility to simulate operational and accidental conditions. It will also improve the user-friendliness towards the users of this program)
- **As a tool of uncertainty qualifications** (This program can also be used as a tool of investigation of how design uncertainties of any components affect the operation dynamics.)
- **Data generative tool** (The autonomous control & system of nuclear power plants can also be trained based on the training data generated by this program.)

Bibliography

[1] Ridoan Karim et al. "Nuclear Energy Development in Bangladesh: A Study of Opportunities and Challenges". In: *Energies* 11 (June 2018), p. 1672. DOI: 10.3390/en11071672.

[2] Ebrahim Modarresi and Faridoon Shabaninia. "Fuzzy Automating Strategies of Emergency Operation for SGTR Accident in VVER 1000 Nuclear Reactor". In: *Universal Journal of Control and Automation* 3 (June 2015), pp. 15–27. DOI: 10.13189/ujca.2015.030201.

[3] Gustavo Boroni and Clausse Alejandro. "Ludwig: A Training Simulator of the Safety Operation of a CANDU Reactor". In: *Science and Technology of Nuclear Installations* 2011 (Jan. 2011). DOI: 10.1155/2011/802410.

[4] Masoud Naghedolfeizi. "Dynamic Modeling of a Pressurized Water Reactor Plant for Diagnostics and Control". In: 1990. URL: https://api.semanticscholar.org/CorpusID:59698783.

[5] Samet E. Arda and Keith E. Holbert. "Nonlinear dynamic modeling and simulation of a passively cooled small modular reactor". In: *Progress in Nuclear Energy* 91 (2016), pp. 116–131. ISSN: 0149-1970. DOI: https://doi.org/10.1016/j.pnucene.2016.03.033. URL: https://www.sciencedirect.com/science/article/pii/S0149197016300798.

[6] Mohamed Rabie Ahmed Ali. "Lumped Parameter, State Variable Dynamic Models for U-tube Recirculation Type Nuclear Steam Generators." In: *PhD diss., University of Tennessee* (1976).

[7] Amir Alramady, Sheikha Al-Sharif, and Sherif Nafee. "Modeling of UTSG in the Pressurized Water Reactor Using Accurate Formulae of Thermodynamic Properties". In: *Journal of Applied Mathematics and Physics* 09 (Jan. 2021), pp. 947–967. DOI: 10.4236/jamp.2021.95065.

[8] Vineet Vajpayee et al. "Dynamic modelling, simulation, and control design of a pressurized water-type nuclear power plant". In: *Nuclear Engineering and Design* 370 (2020), p. 110901. ISSN: 0029-5493. DOI: https://doi.org/10.1016/j.

nucengdes.2020.110901. URL: https://www.sciencedirect.com/science/article/pii/S0029549320303952.

[9] Anu Dutta et al. "A Computer Program for Simulating Transient Behavior in Steam Turbine Stage Pressure of AHWR". In: vol. 2006. July 2006. DOI: 10.1115/ICONE14-89684.

[10] Sheng-Jie Hu and Jie-Sheng Wang. "Dynamic Modeling of Steam Condenser and Design of PI Controller Based on Grey Wolf Optimizer". In: *Mathematical Problems in Engineering* 2015 (Jan. 2015), pp. 1–9. DOI: 10.1155/2015/120975.

[11] Adam Klimanek and Ryszard Bialecki. "On a numerical model of a natural draught wet-cooling tower". In: *Archives of Thermodynamics* 29 (Jan. 2008), pp. 63–72.

[12] Weitong Li, Lei Yu, and Tianhong Yuan. "Modeling Study on the Centrifugal Pump for a Floating Nuclear Power Plant". In: *IOP Conference Series: Earth and Environmental Science* 252 (July 2019), p. 032215. DOI: 10.1088/1755-1315/252/3/032215.

[13] Hong Gao et al. "Transient flow analysis in reactor coolant pump systems during flow coastdown period". In: *Nuclear Engineering and Design* 241 (Feb. 2011), pp. 509–514. DOI: 10.1016/j.nucengdes.2010.09.033.

[14] Mohamed El-Sefy et al. "System dynamics simulation of the thermal dynamic processes in nuclear power plants". In: *Nuclear Engineering and Technology* 51 (Apr. 2019). DOI: 10.1016/j.net.2019.04.017.

[15] Y. Cengel and J. Cimbala. *Fluid Mechanics Fundamentals and Applications: Third Edition*. MCGRAW-HILL US HIGHER ED, 2013. ISBN: 9780077595418. URL: https://books.google.com.bd/books?id=QZIjAAAAQBAJ.

[16] Ian H. Bell et al. "Pure and Pseudo-pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp". In: *Industrial & Engineering Chemistry Research* 53.6 (2014), pp. 2498–2508. DOI: 10.1021/ie4033999. eprint: http://pubs.acs.org/doi/pdf/10.1021/ie4033999. URL: http://pubs.acs.org/doi/abs/10.1021/ie4033999.

[17] Steven C. Chapra and Raymond Canale. *Numerical Methods for Engineers*. 5th ed. USA: McGraw-Hill, Inc., 2005. ISBN: 0073101567.

[18] J.J. Duderstadt and L.J. Hamilton. *Nuclear Reactor Analysis*. Wiley, 1976. ISBN: 9788126541218. URL: https://books.google.com.bd/books?id=xWvsswEACAAJ.

[19] Jiashuang Wan et al. "Dynamic modeling of AP1000 steam generator for control system design and simulation". In: *Annals of Nuclear Energy* 109 (2017), pp. 648–657. ISSN: 0306-4549. DOI: https://doi.org/10.1016/j.anucene.2017.05.016. URL: https://www.sciencedirect.com/science/article/pii/S0306454916303541.

[20] Weitong Li, Lei Yu, and Tianhong Yuan. "Modeling Study on the Centrifugal Pump for a Floating Nuclear Power Plant". In: *IOP Conference Series: Earth and Environmental Science* 252.3 (Apr. 2019), p. 032215. DOI: 10.1088/1755-1315/252/3/032215. URL: https://dx.doi.org/10.1088/1755-1315/252/3/032215.

[21] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[22] Wes McKinney et al. "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[23] Nuclear Power. *Nuclear Power*. 2023. URL: https://www.nuclear-power.com/.

Source Code

```
#########################################################################
                      AZOG  a PWR system analysis program

print('           #                        #       ')
print("          ####                    ####      ")
print('         $$$$$$                  $$$$$$     ')
print('       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ')
print('     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%')
print('    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%')
print('  %%%%%%%%@@@%%%%%%%%%%%%%%%%%%%@@@%%%%%%%%%')
print('%%%%%%%%@@@@@@@@%%%%%%%%%%%%%%%%@@@@@@@@%%%%%%%%')
print('%%%%%%%%@@@@@@%%%%%%%%%%%%%%%%%%%@@@@@@%%%%%%%%%')
print('%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%')
print('%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%')
print(' %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%')
print('  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%')
print('   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%')
print('     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%')
print('       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ')
print('         %%%%%%%%%%%%%%%%%%%%%%%%%%%')
print('             %%%%%%%%%%%%%%%%%%%%%')
print('                  A   Z   O   G')
print('        a PWR system analysis program  \n')
print('PROGRAMMER:')
print("EBNY WALID AHAMMED ")
print("Undergrad Student --Level 4 term 2")
print("Dept of Nuclear Engineering")
print("University of Dhaka")
```

```
 Reactor_core_details.py
 ----------------------------------------------------------------
 import numpy as np
import scipy as sp
from CoolProp.CoolProp import PropsSI

class Reactor_Core():
    def __init__(self,TemperatureFuel:list,TemperatureModerator:list,
    Power:float,Precursor:float,Pressure:float,Temp_hotleg:float
                ,Temp_coldleg:float,Temp_lowerplenum:float,
                Temp_upplerplenum:float):
        self.d=3.62712
        self.core_height=4.38912
        self.CoolantDensity=PropsSI('D','P',Pressure,'Q',0,'water')
        self.active_height=3
        self.df=.95e-2
        self.gas_gap=0.057e-2
        self.N_rods=10693
        self.N_fue_rods=9768
        self.rho_f=10960
        #self.area=np.pi*self.N*self.active_height*self.df**2/4
        self.U=1370

        #group const
        self.Beta1=0.000243
        self.Beta2=0.001363
        self.Beta3=0.001203
        self.Beta4=0.002605
        self.Beta5=0.000829
        self.Beta6=0.000166

        self.total_delayed_const=(self.Beta1+
        self.Beta2+self.Beta3+ self.Beta4+
        self.Beta5+self.Beta6)

        #decay const
        self.Lamda1=0.0127
        self.Lamda2=0.0317
        self.Lamda3=0.115
```

```
self.Lamda4=0.311
self.Lamda5=1.40
self.Lamda6=3.87



self.NGT=3e-5
self.Alpha_m=1.1192*10**-7
self.Alpha_f=6.111*10**-6
self.Alpha_p=-1.8e-6

self.Fr=0.975                    #fission power factor
self.Cpf=1136                    #fuel heat conductivity
self.fuel_pitch=1.26e-2
self.Cpc=PropsSI('C','P',Pressure,'T',
np.mean(TemperatureModerator),'water')   #coolant conductivity
                                 #number of fuel rods
                   #fuel diameter
self.weight_of_a_fuel_rod=2

#volumes
self.CoolantVcore=(self.fuel_pitch**2-np.pi*self.df**2/4)*
self.active_height* self.N_rods
self.area=np.pi*self.df*self.active_height*self.N_fue_rods

self.FuelMass=np.pi*(self.df-2*self.gas_gap)**2
*self.active_height*self.N_fue_rods*self.rho_f/4
self.Wc=1889.68208

"""initial conditions"""
self.NominalPower=Power
self.Precursor=Precursor
self.PowerRatio=1
self.ExternalReactivity=0.0

 #control parameter will be used by control driving program


if len(TemperatureFuel)!=5:
```

```
    raise ValueError("Here should be three initial conditions!")
else:
    self.Tf1=TemperatureFuel[0]
    self.Tf2=TemperatureFuel[1]
    self.Tf3=TemperatureFuel[2]
    self.Tf4=TemperatureFuel[3]
    self.Tf5=TemperatureFuel[4]

if len(TemperatureModerator)!=10:
    raise ValueError('Here should be six initial conditions!')
else:
    self.Tmo1=TemperatureModerator[0]
    self.Tmo2=TemperatureModerator[1]
    self.Tmo3=TemperatureModerator[2]
    self.Tmo4=TemperatureModerator[3]
    self.Tmo5=TemperatureModerator[4]
    self.Tmo6=TemperatureModerator[5]
    self.Tmo7=TemperatureModerator[6]
    self.Tmo8=TemperatureModerator[7]
    self.Tmo9=TemperatureModerator[8]
    self.Tmo10=TemperatureModerator[9]

self.Tcl=Temp_coldleg
self.Thl=Temp_hotleg
self.Tup=Temp_upplerplenum
self.Tlp=Temp_lowerplenum

self.tempmod=np.sum(TemperatureModerator)
self.tempfuel=np.sum(TemperatureFuel)
self.piping_length=37.00756804766576

self.Pressure=Pressure

""" necessary calculations """
"""mass"""
self.core_center_factor=0.05
self.pipe_diameter=850e-3
self.RCP_position_factor=0.5
self.Mcl=((self.piping_length-self.core_height)*
```

```
    self.RCP_position_factor*np.pi*self.pipe_diameter**2/4)*
    PropsSI("D","P",self.Pressure,"Q",0,'water')
    self.Mlp=((self.core_height-self.active_height)*
    self.d**2*PropsSI("D","P",self.Pressure,"Q",0,'water'))*
    self.core_center_factor/4
    self.Mup=((self.core_height-self.active_height)*
    self.d**2*PropsSI("D","T",self.Tup,"Q",0,'water'))
    *(1-self.core_center_factor)/4
    self.Mhl=((self.piping_length-self.core_height)*
    (1-self.RCP_position_factor)*np.pi*self.pipe_diameter**2/4)*
    PropsSI("D","P",self.Pressure,"Q",0,'water')

    '''this values still not known. '''

    self.Mf=self.FuelMass/5
    self.Mmo=self.CoolantVcore*self.CoolantDensity /10



    self.Lamda=self.total_delayed_const/
    (self.Beta1/self.Lamda1+self.Beta2/self.Lamda2+
    self.Beta3/self.Lamda3+
    self.Beta4/self.Lamda4+
    self.Beta5/self.Lamda5+
    self.Beta6/self.Lamda6)

    self.reactivity=0

'''############################################################'''
                                Neutronics an Power
'''############################################################'''

def Reactivity(self,FuelTempSum:float,ModeratorTempSum):
    self.reactivity=self.ExternalReactivity+self.Alpha_f*
    (self.tempfuel-FuelTempSum)/5+self.Alpha_m*
    (self.tempmod-ModeratorTempSum)/10

def DPowerRatio(self):
    dtdPP0=((self.reactivity-self.total_delayed_const)*
```

```
    self.PowerRatio)/self.NGT+
    self.Lamda*self.Precursor
    return dtdPP0

def DPrecoursor(self):
    dtdc=self.total_delayed_const*
    self.PowerRatio/self.NGT-self.Lamda*self.Precursor
    return dtdc
'''############################################################'''
                                        Fuel Region
'''#################################################################'''
def DTf1(self):
    dtdTf1=self.Fr*self.NominalPower*
    self.PowerRatio/(self.Mf*self.Cpf)+
    self.U*self.area*(self.Tmo1-self.Tf1)/(self.Mf*self.Cpf)
    return dtdTf1

def DTf2(self):
    dtdTf2=self.Fr*self.NominalPower*
    self.PowerRatio/(self.Mf*self.Cpf)+self.U*
    self.area*(self.Tmo3-self.Tf2)/(self.Mf*self.Cpf)
    return dtdTf2

def DTf3(self):
    dtdTf3=self.Fr*self.NominalPower*
    self.PowerRatio/(self.Mf*self.Cpf)+
    self.U*self.area*(self.Tmo5-self.Tf3)/(self.Mf*self.Cpf)
    return dtdTf3

def DTf4(self):
    dtdTf2=self.Fr*self.NominalPower*
    self.PowerRatio/(self.Mf*self.Cpf)+self.U*
    self.area*(self.Tmo7-self.Tf4)/(self.Mf*self.Cpf)
    return dtdTf2

def DTf5(self):
    dtdTf3=self.Fr*self.NominalPower*
    self.PowerRatio/(self.Mf*self.Cpf)+self.U*
    self.area*(self.Tmo9-self.Tf5)/(self.Mf*self.Cpf)
```

```
    return dtdTf3
'''#####################################################'''
                            MODERATOR REGION
'''#####################################################'''

def DTmo1(self):

    self.time_constmo=self.Mmo/(self.Wc*2)
    a=(1-self.Fr)*self.NominalPower*
    self.PowerRatio/(self.Mmo*self.Cpc)
    b=self.U*self.area*
    (self.Tf1-self.Tmo1)/(self.Mmo*self.Cpc)+
    (self.Tlp-self.Tmo1)/self.time_constmo

    dtdTm01=a+b
    return dtdTm01

def DTmo2(self):

    self.time_constmo=self.Mmo/(self.Wc*2)
    a=(1-self.Fr)*self.NominalPower*
    self.PowerRatio/(self.Mmo*self.Cpc)
    b=self.U*self.area*(self.Tf1-self.Tmo1)/(self.Mmo*self.Cpc)+
    (self.Tmo1-self.Tmo2)/self.time_constmo

    dtdTmo2=a+b

    return dtdTmo2


def DTmo3(self):

    self.time_constmo=self.Mmo/(self.Wc*2)
    a=(1-self.Fr)*self.NominalPower*
    self.PowerRatio/(self.Mmo*self.Cpc)
    b=self.U*self.area*
    (self.Tf2-self.Tmo3)/(self.Mmo*self.Cpc)+
    (self.Tmo2-self.Tmo3)/self.time_constmo
    dtdTm03=a+b
```

```
    return dtdTm03


def DTmo4(self):
    self.time_constmo=self.Mmo/(self.Wc*2)
    a=(1-self.Fr)*
    self.NominalPower*self.PowerRatio/(self.Mmo*self.Cpc)
    b=self.U*self.area*
    (self.Tf2-self.Tmo3)/(self.Mmo*self.Cpc)+
    (self.Tmo3-self.Tmo4)/self.time_constmo
    dtdTm04=a+b
    return dtdTm04

def DTmo5(self):
    self.time_constmo=self.Mmo/(self.Wc*2)
    a=(1-self.Fr)*self.NominalPower*
    self.PowerRatio/(self.Mmo*self.Cpc)
    b=self.U*self.area*(self.Tf3-self.Tmo5)/(self.Mmo*self.Cpc)+
    (self.Tmo4-self.Tmo5)/self.time_constmo
    dtdTm05=a+b
    return dtdTm05

def DTmo6(self):

    self.time_constmo=self.Mmo/(self.Wc*2)
    a=(1-self.Fr)*self.NominalPower*
    self.PowerRatio/(self.Mmo*self.Cpc)
    b=self.U*self.area*
    (self.Tf3-self.Tmo5)/(self.Mmo*self.Cpc)+
    (self.Tmo5-self.Tmo6)/self.time_constmo

    dtdTmo2=a+b

    return dtdTmo2


def DTmo7(self):

    self.time_constmo=self.Mmo/(self.Wc*2)
```

```
        a=(1-self.Fr)*self.NominalPower*
        self.PowerRatio/(self.Mmo*self.Cpc)
        b=(self.Tmo6-self.Tmo7)/self.time_constmo+
        self.U*self.area*
        (self.Tf4-self.Tmo7)/(self.Mmo*self.Cpc)
        dtdTm03=a+b
        return dtdTm03


    def DTmo8(self):
        self.time_constmo=self.Mmo/(self.Wc*2)
        a=(1-self.Fr)*self.NominalPower*
        self.PowerRatio/(self.Mmo*self.Cpc)
        b=self.U*self.area*
        (self.Tf4-self.Tmo7)/(self.Mmo*self.Cpc)+
        (self.Tmo7-self.Tmo8)/self.time_constmo
        dtdTm04=a+b
        return dtdTm04

    def DTmo9(self):
        self.time_constmo=self.Mmo/(self.Wc*2)
        a=(1-self.Fr)*self.NominalPower*
        self.PowerRatio/(self.Mmo*self.Cpc)
        b=self.U*self.area*
        (self.Tf5-self.Tmo9)/(self.Mmo*self.Cpc)+
        (self.Tmo8-self.Tmo9)/self.time_constmo
        dtdTm05=a+b
        return dtdTm05

    def DTmo10(self):
        self.time_constmo=self.Mmo/(self.Wc*2)
        a=(1-self.Fr)*self.NominalPower*
        self.PowerRatio/(self.Mmo*self.Cpc)
        b=self.U*self.area*
        (self.Tf5-self.Tmo9)/(self.Mmo*self.Cpc)+
        (self.Tmo9-self.Tmo10)/self.time_constmo
        dtdTm06=a+b
        return dtdTm06
```

```
def DTcl(self,Temp_RCP:float):

    dtdTcl=self.Wc*(Temp_RCP-self.Tcl)/self.Mcl
    return dtdTcl

def DThl(self):

    dtdTcl=self.Wc*(self.Tup-self.Thl)/self.Mhl
    return dtdTcl

def DTup(self):
    self.Mup=((self.core_height-self.active_height)*
    self.d**2*PropsSI("D","P",self.Pressure,"Q",0,'water'))*
    (1-self.core_center_factor)/4
    dtdTcl=self.Wc*(self.Tmo10-self.Tup)/self.Mup
    return dtdTcl

def DTlp(self):
    self.Mlp=((self.core_height-self.active_height)*self.d**2*
    PropsSI("D","P",self.Pressure,"Q",0,'water'))*
    self.core_center_factor/4
    dtdTcl=self.Wc*(self.Tcl-self.Tlp)/self.Mlp
    return dtdTcl


def integrator(self,function,
argsforfunction:list,intitial_cond,time_step):
    l=len(argsforfunction)

    if l==0:
        return function()*time_step+intitial_cond
    elif l==1:
        arg1=argsforfunction[0]
        return function(arg1)*time_step+intitial_cond
    elif l==2:
        arg1=argsforfunction[0]
        arg2=argsforfunction[1]
        return function(arg1,arg2)*time_step+intitial_cond
    elif l==3:
```

```
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            arg3=argsforfunction[2]
            return function(arg1,arg2,arg3)*time_step+
            intitial_cond
        else:
            raise   AttributeError("agrs in your differential
            function were not correct! Fix them")


'''------------------------- Done------------------------------ '''

 ---------------------------------------------------------------
 Reactor_core_model_B.py
 ---------------------------------------------------------------

import numpy as np
from CoolProp.CoolProp import PropsSI
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt
from tqdm import tqdm

class CoreInside():
    def __init__(self,TemperatureFuel:list,
    TemperatureModerator:list,Power:float,
    Precursor:float,Pressure:float):

        self.diameter=3.62712
        self.core_height=4.38912
        self.CoolantDensity=PropsSI('D','P',Pressure,'Q',0,'water')
        self.area=14855.196091203/3
        self.h=1134      #needs more exect value conductivity

        #group const
        self.Beta1=0.000243
        self.Beta2=0.001363
        self.Beta3=0.001203
        self.Beta4=0.002605
        self.Beta5=0.000829
```

```
self.Beta6=0.000166

self.total_delayed_const=(self.Beta1+
self.Beta2+self.Beta3+
self.Beta4+self.Beta5+self.Beta6)

#decay const
self.Lamda1=0.0127
self.Lamda2=0.0317
self.Lamda3=0.115
self.Lamda4=0.311
self.Lamda5=1.40
self.Lamda6=3.87

self.NGT=2*10**-5
self.Alpha_m=-1.1192*10**-7
self.Alpha_f=-6.111*10**-6

self.Fr=0.974       #fission power factor
self.Cpc=PropsSI('C','P',Pressure,'Q',0,'water')
#coolant conductivity
self.Cpf=1136
#fuel heat conductivity
#volumes
self.CoolantVcore=15.2911

self.FuelMass=101032.711
self.Wc=18899.68208

"""initial conditions"""
self.NominalPower=Power
self.Precursor=Precursor
self.PowerRatio=1
self.ExternalReactivity=0.00000

 #control parameter will be used by
 control driving program
```

```
    if len(TemperatureFuel)!=3:
        raise ValueError("Here should be
        three initial conditions!")
    else:
        self.Tf1=TemperatureFuel[0]
        self.Tf2=TemperatureFuel[1]
        self.Tf3=TemperatureFuel[2]

    if len(TemperatureModerator)!=6:
        raise ValueError('Here should be
        six initial conditions!')
    else:
        self.Tmo1=TemperatureModerator[0]
        self.Tmo2=TemperatureModerator[1]
        self.Tmo3=TemperatureModerator[2]
        self.Tmo4=TemperatureModerator[3]
        self.Tmo5=TemperatureModerator[4]
        self.Tmo6=TemperatureModerator[5]

    self.tempmod=np.sum(TemperatureModerator)
    self.tempfuel=np.sum(TemperatureFuel)
    """ necessary calculations """
    """mass"""
    self.Mf=self.FuelMass/3 #as there will three lumps
    self.Mmo=self.CoolantVcore*self.CoolantDensity /3

    """time const"""
    self.time_constmo=self.Mmo/(self.Wc*2)

    self.Lamda=self.total_delayed_const/(self.Beta1/self.Lamda1+
    self.Beta2/self.Lamda2+self.Beta3/self.Lamda3+
    self.Beta4/self.Lamda4\self.Beta5/self.Lamda5+
    self.Beta6/self.Lamda6)

    self.reactivity=0
def Reacivity(self,FuelTempSum:float,
ModeratorTempSum):

    self.reactivity=self.ExternalReactivity+
```

```
        self.Alpha_f*(self.tempfuel-FuelTempSum)/3+
        self.Alpha_m*(self.tempmod-ModeratorTempSum)/6

    def DPowerRatio(self):
        dtdPP0=((self.reactivity-self.total_delayed_const)*
        self.PowerRatio)/self.NGT+
        self.Lamda*self.Precursor
        return dtdPP0

    def DPrecoursor(self):
        dtdc=self.total_delayed_const*
        self.PowerRatio/self.NGT-self.Lamda*self.Precursor
        return dtdc

    def DTf1(self):
        dtdTf1=self.Fr*self.NominalPower*
        self.PowerRatio/(self.Mf*self.Cpf)+
        self.h*self.area*(self.Tmo1-self.Tf1)/(self.Mf*self.Cpf)
        return dtdTf1

    def DTf2(self):
        dtdTf2=self.Fr*self.NominalPower*
        self.PowerRatio/(self.Mf*self.Cpf)+
        self.h*self.area*(self.Tmo3-self.Tf2)/(self.Mf*self.Cpf)
        return dtdTf2

    def DTf3(self):
        dtdTf3=self.Fr*self.NominalPower*
        self.PowerRatio/(self.Mf*self.Cpf)+
        self.h*self.area*(self.Tmo5-self.Tf3)/(self.Mf*self.Cpf)
        return dtdTf3

    def DTm01(self,LowerPlenum:object):
        self.time_constmo=self.Mmo/(self.Wc*2)
        dtdTm01=(1-self.Fr)*self.NominalPower*
        self.PowerRatio/(self.Mmo*self.Cpc)+
            self.h*self.area*(self.Tf1-self.Tmo1)/(self.Mmo*self.Cpc)+
            (LowerPlenum.Tlp-self.Tmo1)/self.time_constmo
        return dtdTm01
```

```
def DTm02(self):
    self.time_constmo=self.Mmo/(self.Wc*2)
    dtdTm02=(1-self.Fr)*self.NominalPower*
    self.PowerRatio/(self.Mmo*self.Cpc)+\
        self.h*self.area*
        (self.Tf1-self.Tmo1)/(self.Mmo*self.Cpc)+
        (self.Tmo1-self.Tmo2)/self.time_constmo

    return dtdTm02


def DTm03(self):
    self.time_constmo=self.Mmo/(self.Wc*2)
    dtdTm03=(1-self.Fr)*self.NominalPower*
    self.PowerRatio/(self.Mmo*self.Cpc)+
    self.h*self.area*
    (self.Tf2-self.Tmo3)/(self.Mmo*self.Cpc)\
        +(self.Tmo2-self.Tmo3)/self.time_constmo

    return dtdTm03


def DTm04(self):
    self.time_constmo=self.Mmo/(self.Wc*2)
    dtdTm04=(1-self.Fr)*self.NominalPower*
    self.PowerRatio/(self.Mmo*self.Cpc)+\
        self.h*self.area*
        (self.Tf2-self.Tmo3)/(self.Mmo*self.Cpc)+
        (self.Tmo3-self.Tmo4)/self.time_constmo
    return dtdTm04

def DTm05(self):
    self.time_constmo=self.Mmo/(self.Wc*2)
    dtdTm05=(1-self.Fr)*self.NominalPower*
    self.PowerRatio/(self.Mmo*self.Cpc)+\
        self.h*self.area*
        (self.Tf3-self.Tmo5)/(self.Mmo*self.Cpc)+
        (self.Tmo4-self.Tmo5)/self.time_constmo
```

```
        return dtdTm05

    def DTm06(self):
        self.time_constmo=self.Mmo/(self.Wc*2)
        dtdTm06=(1-self.Fr)*self.NominalPower*
        self.PowerRatio/(self.Mmo*self.Cpc)+\
            self.h*self.area*
            (self.Tf3-self.Tmo5)/(self.Mmo*self.Cpc)+
            (self.Tmo5-self.Tmo6)/self.time_constmo
        return dtdTm06

    def integrator(self,function,argsforfunction:list,
    intitial_cond,time_step):
        l=len(argsforfunction)

        if l==0:
            return function()*time_step+intitial_cond
        elif l==1:
            arg1=argsforfunction[0]
            return function(arg1)*time_step+intitial_cond
        elif l==2:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            return function(arg1,arg2)*time_step+intitial_cond
        elif l==3:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            arg3=argsforfunction[2]
            return function(arg1,arg2,arg3)*
            time_step+intitial_cond
        else:
            raise   AttributeError("agrs in your
            differential function were not correct! Fix them")

class Hotleg():
    def __init__(self,TempHotLeg:float):
        self.Thl=TempHotLeg
        self.CoolantVhl=28.3168
        self.Wc=18899.68208
```

```
        self.CoolantDensity=732.18278
        self.Mhl=self.CoolantVhl*self.CoolantDensity
        self.time_consthl=self.Mhl/self.Wc

    def DThl(self,UpperPlenum:object):

        dtdThl=(UpperPlenum.Tup-self.Thl)/self.time_consthl
        return dtdThl

    def integrator(self,function,argsforfunction:list,
    intitial_cond,time_step):
        l=len(argsforfunction)

        if l==0:
            return function()*time_step+intitial_cond
        elif l==1:
            arg1=argsforfunction[0]
            return function(arg1)*time_step+intitial_cond
        elif l==2:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            return function(arg1,arg2)*time_step+intitial_cond
        elif l==3:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            arg3=argsforfunction[2]
            return function(arg1,arg2,arg3)*time_step+
            intitial_cond
        else:
            raise   AttributeError("agrs in your differential
            function were not correct! Fix them")


class ColdLeg():
    def __init__(self,TempColdLeg:float) -> None:
        self.Tcl=TempColdLeg
        self.Wc=18899.68208
        self.CoolantDensity=732.18278
        self.CoolantVcl=56.63369
```

```
        self.Mcl=self.CoolantVcl*self.CoolantDensity
        self.time_constcl=self.Mcl/self.Wc

    def DTcl(self,TempRCPexit:float):
        """temperature exit will accessed through
        objects. but right now it's initilized as
        a float data type """
        dtdTcl=(TempRCPexit-self.Tcl)/self.time_constcl
        return dtdTcl

    def integrator(self,function,argsforfunction:list,
    intitial_cond,time_step):
        l=len(argsforfunction)

        if l==0:
            return function()*time_step+intitial_cond
        elif l==1:
            arg1=argsforfunction[0]
            return function(arg1)*time_step+intitial_cond
        elif l==2:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            return function(arg1,arg2)*time_step+intitial_cond
        elif l==3:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            arg3=argsforfunction[2]
            return function(arg1,arg2,arg3)*time_step+
            intitial_cond
        else:
            raise   AttributeError("agrs in your differential
            function were not correct! Fix them")

class UpperPlenum():
    def __init__(self,TempUpperPlenum:float):
        self.Tup=TempUpperPlenum
        self.Wc=18899.68208
        self.CoolantDensity=732.18278
        self.CoolantVup=38.9813
```

```
        self.Mup=self.CoolantDensity*self.CoolantVup
        self.time_constup=self.Mup/self.Wc

    def DTup(self,ReactorCoreInside:object):
        dtdTup=(ReactorCoreInside.Tmo6-self.Tup)/self.time_constup
        return dtdTup
    def integrator(self,function,argsforfunction:list,
    intitial_cond,time_step):
        l=len(argsforfunction)

        if l==0:
            return function()*time_step+intitial_cond
        elif l==1:
            arg1=argsforfunction[0]
            return function(arg1)*time_step+intitial_cond
        elif l==2:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            return function(arg1,arg2)*time_step+intitial_cond
        elif l==3:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            arg3=argsforfunction[2]
            return function(arg1,arg2,arg3)*time_step+
            intitial_cond
        else:
            raise   AttributeError("agrs in your differential
            function were not correct! Fix them")



class LowerPlenum():
    def __init__(self,TempLowerPlenum:float):
        self.Tlp=TempLowerPlenum
        self.Wc=18899.68208
        self.CoolantDensity=732.18278
        self.CoolantVlp=50.7091
        self.Mlp=self.CoolantVlp*self.CoolantDensity
        self.time_constlp=self.Mlp/self.Wc
```

```
    def DTlp(self,ColdLeg:object):
        dtdTlp=(ColdLeg.Tcl-self.Tlp)/self.time_constlp
        return dtdTlp

    def integrator(self,function,argsforfunction:list,
    intitial_cond,time_step):
        l=len(argsforfunction)

        if l==0:
            return function()*time_step+intitial_cond
        elif l==1:
            arg1=argsforfunction[0]
            return function(arg1)*time_step+intitial_cond
        elif l==2:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            return function(arg1,arg2)*time_step+intitial_cond
        elif l==3:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            arg3=argsforfunction[2]
            return function(arg1,arg2,arg3)*time_step+
            intitial_cond
        else:
            raise   AttributeError("agrs in your differential
            function were not correct! Fix them")
```

```
 ----------------------------------------------------------------
 pressurizer.py
 ----------------------------------------------------------------

 import numpy as np
from CoolProp.CoolProp import PropsSI
from scipy.interpolate import interp1d

class Pressurizer():
    def __init__(self,pressurizer_diameter:float,
    Pressure:float,Temp_coldleg:float):
```

```
self.Pressure=Pressure
self.diameter=pressurizer_diameter
self.l=14.2524
self.l_w=8.5527
self.J_p=5.4027
self.hw_bar=9.7209e5

''' m_spray and Q automatic variable and
depends on the operation. I will left this
on the user to denfine when to operate them.
'''
self.m_spray=0
self.Q=0
self.area=self.diameter**2*np.pi/4
'''
-----------------------------------------------------
where the index j = 1 to N represent coolant nodes in the
following order, lower plenum, coolant node 1 and 2, upper
plenum, hot-leg, inlet plenum, PCL 1 and 2 and outlet plenum,
and cold-leg. Based on the surge  will be calculated
-----------------------------------------------------
'''
self.V_times_nu=np.array([.5991,0.1814,0.1814,
1.3164,1.3164,0.2752,0.277,0.277,0.022,
0.6022,0.6022,0.2776,0.2776,0.1927])

''' Q--->  0  ----> represents water
    Q--->  1  ----> represents steam
'''
self.rho_w=PropsSI("D",'P',self.Pressure,'Q',0,'water')
self.rho_s=PropsSI("D","P",self.Pressure,'Q',1,'water')
self.Nu_w=1/PropsSI("D",'P',self.Pressure,'Q',0,'water')
self.Nu_s=1/PropsSI("D",'P',self.Pressure,'Q',1,'water')
self.h_w=PropsSI("H",'P',self.Pressure,'Q',0,'water')
self.h_spr=PropsSI("H","T",Temp_coldleg,'Q',0,'water')

'''---------------partial derivative co-efficient--------'''
self.k1p=PropsSI('d(D)/d(T)|P','P',self.Pressure,'Q',0,'Water')
```

```
        self.k2p=PropsSI('d(D)/d(T)|P','P',self.Pressure,'Q',1,'Water')
        self.k3p=PropsSI('d(H)/d(T)|P','P',self.Pressure,'Q',1,'Water')

        '''----------  calculation of partital derivative  -----'''

        Pressure_=np.logspace(3,16.909,100000,base=np.e)

        density_=np.array(PropsSI("D",'P',Pressure_,'Q',0,'water'))
        nu_s=1/density_
        delta_nu_s=np.diff(nu_s)
        delta_pressure=np.diff(Pressure_)
        print(len(delta_nu_s),len(delta_pressure),len(Pressure_))
        self.k4p=interp1d(Pressure_[:len(Pressure_)-1],
        delta_nu_s/delta_pressure)

        '''------------------------------------------------ '''

    def DPp(self,DTemp_list:list,Temp_coldleg):

        '''---------------surge calculation----------'''
        if len(DTemp_list)!=14:
            raise ValueError('You need all the temperature derivative in the core:')
        else:
            m_surge=np.sum(self.V_times_nu*np.array(DTemp_list))

        nu_s=1/PropsSI("D","P",self.Pressure,'Q',1,'water')
        h_spr=PropsSI("H","T",Temp_coldleg,'Q',0,'water')
        h_w=PropsSI("H",'P',self.Pressure,'Q',0,'water')
        rho_w=PropsSI("D",'P',self.Pressure,'Q',0,'water')
        rho_s=PropsSI("D","P",self.Pressure,'Q',1,'water')

        C1p=(PropsSI("D",'P',self.Pressure,'Q',0,'water')/
        PropsSI("D","P",self.Pressure,'Q',1,'water'))-1

        k3p=PropsSI('d(H)/d(T)|P','P',self.Pressure,'Q',1,'Water')
        k4p=self.k4p(self.Pressure) #interpolated function
        Vw=self.area*self.l_w
        Vs=self.area*(self.l-self.l_w)
```

```
    a=self.Q+m_surge*((self.Pressure*nu_s)/(self.J_p*C1p))
    b=self.m_spray*(h_spr-h_w+(self.hw_bar/C1p)+
    (self.Pressure/(rho_w*self.J_p*C1p)))
    c=Vw*rho_w*(k3p+(k4p*self.Pressure)/self.J_p)+
    (Vs*rho_s*k4p*self.Pressure-Vw)/self.J_p
    d=(self.hw_bar+self.Pressure*nu_s/self.J_p)

    dtdPp=(a+b)/(c+d)
    return dtdPp

def Dlw(self,DTemp_list:list,Temp_coldleg):
    rho_s=PropsSI("D","P",self.Pressure,'Q',1,'water')

    '''------------------surge calculation----------'''
    if len(DTemp_list)!=14:
        raise ValueError('You need all the
        temperature derivative in the core:')
    else:
        m_surge=np.sum(self.V_times_nu*np.array(DTemp_list))

    k1p=PropsSI('d(D)/d(T)|P','P',self.Pressure,'Q',0,'Water')
    k2p=PropsSI('d(D)/d(T)|P','P',self.Pressure,'Q',1,'Water')
    C1p=(PropsSI("D",'P',self.Pressure,'Q',0,'water')/
    PropsSI("D","P",self.Pressure,'Q',1,'water'))-1
    C2p=self.area*(self.l-self.l_w)*(C1p+1)*k2p+
    self.area*self.l_w*k1p

    a=(self.area*(self.l-self.l_w)*k2p-C1p/C2p)*
    self.DPp(DTemp_list,Temp_coldleg)
    b=(m_surge/C1p)+
    ((C2p*self.DPp(DTemp_list,Temp_coldleg))-
    m_surge-self.m_spray)/(C1p)**2
    dtdlw=(a+b)/rho_s*self.area

    return dtdlw

def integrator(self,function,argsforfunction:list,
intitial_cond,time_step):
    l=len(argsforfunction)
```

```
        if l==0:
            return function()*time_step+intitial_cond
        elif l==1:
            arg1=argsforfunction[0]
            return function(arg1)*time_step+intitial_cond
        elif l==2:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            return function(arg1,arg2)*time_step+intitial_cond
        elif l==3:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            arg3=argsforfunction[2]
            return function(arg1,arg2,arg3)*time_step+
            intitial_cond
        else:
            raise   AttributeError("agrs in your differential
            function were not correct! Fix them")
```

```
 ---------------------------------------------------------------------
 u_tube_steam_generator.py
 ---------------------------------------------------------------------
import numpy as np
import scipy as sp
from CoolProp.CoolProp import PropsSI

"""    --------------- Model Begin -------------------"""

class u_tube_steam_generator():
    def __init__(self,primary_coolant_inlet_temperature:float,
    primary_coolant_outlet_temperature:float,
    feed_water_inlet_temperature:float,
    drum_water_temp:float,
    saturation_temp:float,down_comer_temp:float,
    feed_water_flow_rate:float,PrimaryLumpTemperature:list,
    MetalLumpTemperature:list,Reactor_Pressure:float,
    Steam_pressure:float):
```

```
'''--------  design parameters   ---------'''
self.N=3388
self.L=10.83
self.L_w=3.057   #sub cool region height
self.Ldw=10.83   #Drum water level height
self.R_in=0.0098425 #needs to be checked
self.R_out=0.0111125  #needs to be checked
self.k=15
self.Pressure_r=Reactor_Pressure
self.Pressure_s=Steam_pressure
self.Cl=80e-6
'''-----------area and volume   -------'''
self.P_r1=2*np.pi*self.R_in
self.P_r2=2*np.pi*self.R_out
self.A_r1=np.pi*self.R_in**2
self.A_r2=np.pi*self.R_out**2
self.Ap=self.N*np.pi*self.R_in**2
self.Afs=5.63643
self.Ad=9.39528444
self.rho_m=8050                 #needs the confirmation
self.rho_b=PropsSI('D',"P",self.Pressure_s,'Q',0.99/2,'water')
self.Vp=30.5
self.Vr=13.2523
self.Vdr=124.55748301
self.Vpi=(0.5*self.Vp-self.Ap*self.L)

'''---- coolant and metal conductivity ---------'''
if len(PrimaryLumpTemperature)!=4:
    raise ValueError("Initial condition error!")
else:
    self.Tp1=PrimaryLumpTemperature[0]
    self.Tp2=PrimaryLumpTemperature[1]
    self.Tp3=PrimaryLumpTemperature[2]
    self.Tp4=PrimaryLumpTemperature[3]

if len(MetalLumpTemperature)!=4:
    raise ValueError(" Initial condition error!")
else:
    self.Tm1=MetalLumpTemperature[0]
```

```
    self.Tm2=MetalLumpTemperature[1]
    self.Tm3=MetalLumpTemperature[2]
    self.Tm4=MetalLumpTemperature[3]

self.Tfi=feed_water_inlet_temperature
self.Tpi=primary_coolant_inlet_temperature
self.Tpo=primary_coolant_outlet_temperature
self.Ts1=(saturation_temp+down_comer_temp)/2
self.Tsat=saturation_temp
self.Td=down_comer_temp
self.Tdw=drum_water_temp

self.Cp1=PropsSI("C","T",self.Tp1,'Q',0,'water')
#reactor pressure instead of the Q= 0
self.Cm=460

'''----------      flow rates             ----------'''
self.Win=4950
self.Wp1=self.Win
self.Wpout=self.Wp1
'''------------------secondary loop-------------'''
self.W1=892         #not sure!
self.W2=self.W1
self.W3=self.W1
self.W4=self.W1
self.Wfi=feed_water_flow_rate
self.rho_r=7.94*16.01844634
self.h_b=PropsSI('H','P',self.Pressure_s,'Q',0.4,'water')
'''---------      mass calculation        ---------'''
self.mpi=PropsSI('D','T',self.Tpi,'P',self.Pressure_r,'water')*
self.Vpi #reactor pressure instead of the Q= 0
self.mp1=PropsSI('D','T',self.Tp1,'P',self.Pressure_r,'water')*
self.A_r1*self.L_w #reactor pressure instead of the Q= 0
self.mp2=PropsSI('D','T',self.Tp2,'P',self.Pressure_r,'water')*
self.A_r1*(self.L-self.L_w) #reactor pressure instead of the Q= 0
self.mp3=PropsSI('D','T',self.Tp3,'P',self.Pressure_r,'water')*
self.A_r1*(self.L-self.L_w)
self.mp4=PropsSI('D','T',self.Tp4,'P',self.Pressure_r,'water')*
self.A_r1*(self.L_w)
```

```
        self.mm1=self.N*np.pi*self.L_w*
        (self.R_out**2-self.R_in**2)*self.rho_m
        self.mm2=self.N*np.pi*(self.L-self.L_w)*
        (self.R_out**2-self.R_in**2)*self.rho_m

        self.hi=7.098
        self.hd=4.974
        self.hb=10.618

        self.Upm=1/((1/self.hi)+
        (self.R_in/self.k)*(np.log10((self.R_out+self.R_in)/((self.R_in*2)))))
        self.Ums1=1/((1/self.hd)+
        (self.R_out/self.k)*(np.log10(2*self.R_out/(self.R_out+self.R_in))))
        self.Ums2=1/((1/self.hb)+
        (self.R_out/self.k)*(np.log10(2*self.R_out/(self.R_out+self.R_in))))


'''-------------------------primary loop begin -------------------'''

def DTpi(self,Hot_leg_temp:float):

        dtdTpi=self.Win*(Hot_leg_temp-self.Tpi)/self.mpi
        return dtdTpi

def DTpo(self):
        mpi=PropsSI('D','T',self.Tpi,'P',self.Pressure_r,'water')*
        self.Vpi
        dtdTpo=(self.Tp4-self.Tpo)*self.Win/(mpi)
        return dtdTpo

def DLs1(self):
        rho_s1=PropsSI('D','T',self.Ts1,"P",self.Pressure_s,'water')
        dtdLs1=(self.W1-self.W2)/(rho_s1*self.Afs)
        return dtdLs1

def DTp1(self):

        rho_p=PropsSI('D','T',self.Tp1,'P',self.Pressure_r,'water')
```

```
    Cp1=PropsSI('C','T',self.Tp1,'P',self.Pressure_r,'water')
    mp1=PropsSI('D','T',self.Tp1,'P',self.Pressure_r,'water')*
    self.A_r1*(self.L_w)

    a=self.Win*(self.Tpi-self.Tp1)/(rho_p*self.Ap*self.L_w)
    b=self.Upm*self.P_r1*self.L_w*self.N*(self.Tm1-self.Tp1)
    c=mp1*Cp1

    dtdTp1=a+b/c
    return dtdTp1

def DTp2(self):

    rho_p=PropsSI('D','T',self.Tp4,'P',self.Pressure_r,'water')
    Cp1=PropsSI('C','T',self.Tp4,'P',self.Pressure_r,'water')
    mp2=PropsSI('D','T',self.Tp4,'P',self.Pressure_r,'water')*
    self.A_r1*(self.L-self.L_w)



    a=self.Win*(self.Tp1-self.Tp2)/mp2
    b=-(self.Upm*self.P_r1*self.N*(self.L-self.L_w)*self.N*
    (self.Tp2-self.Tm2))/(mp2*Cp1)
    c=-((self.Tp1-self.Tp2))/(self.L-self.L_w)
    d=self.DLs1()

    dtdTp2=a+b+c*d

    return dtdTp2

def DTp3(self):

    rho_p=PropsSI('D','T',self.Tp4,'P',self.Pressure_r,'water')
    Cp1=PropsSI('C','T',self.Tp4,'P',self.Pressure_r,'water')
    mp3=PropsSI('D','T',self.Tp4,'P',self.Pressure_r,'water')*
    self.A_r1*(self.L-self.L_w)

    a=self.Win*(self.Tp2-self.Tp3)/(rho_p*self.Ap*(self.L-self.L_w))
    b=self.Upm*self.P_r1*self.N*(self.L-self.L_w)*(self.Tm3-self.Tp3)
```

```
    c=mp3*Cp1

    dtdTp3=a+b/c
    return dtdTp3


def DTp4(self):

    rho_p=PropsSI('D','T',self.Tp4,'P',self.Pressure_r,'water')
    Cp1=PropsSI('C','T',self.Tp4,'P',self.Pressure_r,'water')
    mp4=PropsSI('D','T',self.Tp4,'P',self.Pressure_r,'water')*
    self.A_r1*(self.L_w)

    a=self.Win*(self.Tp3-self.Tp4)/(rho_p*self.Ap*self.L_w)
    b=self.Upm*self.P_r1*self.N*(self.L_w)*
    (self.Tm4-self.Tp4)/(mp4*Cp1)
    c=(self.Tp3-self.Tp4)/self.L_w
    dtdTp4=a+b+c*self.DLs1()
    return dtdTp4

'''----------------primary loop done -----------------'''



'''---------------------- Metal lump begin ---------------'''

def DTm1(self):

    a=(self.Upm*self.N*self.P_r1*self.L_w*
    (self.Tp1-self.Tm1)-self.Ums1*self.N*self.P_r2*self.L_w*
    (self.Tm1-(self.Td+self.Tsat)/2))/(self.mm1*self.Cm)
    b=-(self.Tm1-self.Tm2)*0.5/self.L_w
    c=self.DLs1()
    dtdTm1=a+b*c
    return dtdTm1

def DTm2(self):

    a=self.Upm*self.N*self.P_r1*(self.L-self.L_w)*
```

```
        self.Tp2/(self.mm2*self.Cm)
        b=-(self.Upm*self.N*self.P_r1*(self.L-self.L_w)+
        self.Ums2*self.N*self.P_r2*(self.L-self.L_w))*
        self.Tm2/(self.mm2*self.Cm)
        c=self.Ums2*self.N*self.P_r2*(self.L-self.L_w)*
        self.Tsat/(self.mm2*self.Cm)
        d=(self.Tm2-self.Tm1)/(2*(self.L-self.L_w))

        dtdTm2=a+b+c+d*self.DLs1()

        return dtdTm2

    def DTm3(self):

        a=self.Upm*self.N*self.P_r1*(self.L-self.L_w)*
        self.Tp2/(self.mm2*self.Cm)
        b=-(self.Upm*self.N*self.P_r1*(self.L-self.L_w)+
        self.Ums2*self.N*self.P_r2*(self.L-self.L_w))*
        self.Tm3/(self.mm2*self.Cm)
        c=self.Ums2*self.N*self.P_r2*(self.L-self.L_w)*
        self.Tsat/(self.mm2*self.Cm)
        d=(self.Tm3-self.Tm4)/(2*(self.L-self.L_w))

        dtdTm3=a+b+c+d*self.DLs1()

        return dtdTm3

    def DTm4(self):

        a=self.Upm*self.N*self.P_r1*(self.L_w)*
        self.Tp4/(self.mm1*self.Cm)
        b=-(self.Upm*self.N*self.P_r1*(self.L_w)+
        self.Ums1*self.N*self.P_r2*(self.L_w))*self.Tm4/(self.mm1*self.Cm)
        c=-self.Ums1*self.N*self.P_r2*(self.L_w)*
        self.Ts1/(self.mm1*self.Cm)
        d=((self.Tm3-self.Tm4)/(2*(self.L_w)))*self.DLs1()

        dtdTm4=a+b+c+d
        return dtdTm4
```

```
'''---------------- Metal lump done -----------------'''


''' ----------------- secondary lump--------------'''

def DTsat(self):
    self.Ts1=(self.Td+self.Tsat)/2
    Cp2=PropsSI("C","T",self.Ts1,'P',self.Pressure_s,'water')
    rho_s1=PropsSI("D",'T',self.Ts1,'P',self.Pressure_s,'water')
    self.W2=self.W1-self.DLs1()*rho_s1*self.Ap
    a=self.Ums1*self.N*self.P_r2*
    self.L_w*(self.Tm1+self.Tm4-2*self.Ts1)+self.W1*Cp2*
    self.Td-self.W2*Cp2*self.Tsat
    a11=rho_s1*self.Afs*(self.Td+self.Tsat)*self.DLs1()/2
    a22=0.5*rho_s1*self.Afs*Cp2*self.L_w*self.DTd()
    b=0.5*self.Afs*rho_s1*self.L_w*Cp2

    dtdTs1=(a-a11-a22)/b
    return dtdTs1

def Drho_b(self):

    a=(self.W2-self.W3)/(self.Afs*(self.L-self.L_w))
    b=self.rho_b*self.DLs1()/(self.L-self.L_w)

    dtdrho_b=a+b
    return dtdrho_b

def Dh_b(self):
    a=self.Ums2*self.N*self.P_r2*
    (self.L-self.L_w)*(self.Tm2-self.Tsat)+
    self.Ums2*self.N*self.P_r2*(self.L-self.L_w)*
    (self.Tm3-self.Tsat)
    b=self.W2*
    PropsSI('H','T',self.Ts1,'Q',0,'water')-self.W3*
    PropsSI('H','P',self.Pressure_s,'T',self.Ts1,'water')
    c=self.rho_b*self.Afs*self.hb*self.DLs1()
```

```
        f=(self.L-self.L_w)*self.hb*self.Drho_b()*self.Afs
        d=self.rho_b*self.Afs*(self.L-self.L_w)

        dtdh_b=(a+b+c-f)/d
        return dtdh_b

    def Drho_r(self):
        dtdrho_r=(self.W3-self.W2)/self.Vr
        return dtdrho_r

    def DLdw(self):
        Xe=0.2
        a=-self.W1-(1-Xe)*self.W3+self.Wfi
        rho_d=PropsSI('D',"T",self.Tfi,'Q',0,'water')
        dtdLdw=a/(rho_d*self.Ad)
        return dtdLdw

    def DTdw(self):
        Xe=0.2
        self.rho_g=PropsSI("D",' T',self.Tdw,"Q",Xe,'water')
        rho_d=PropsSI('D',"T",self.Tfi,'Q',0,'water')
        dtdTdw=(self.Wfi*self.Tfi-(1-Xe)*
        self.W4*self.Tsat-self.W1*self.Tdw-rho_d*self.Ad*self.DLdw())/
        (self.Ad*rho_d*self.Ldw)
        return dtdTdw

    def Drho_g(self):
        Xe=0.2
        self.rho_g=PropsSI("D",' T',self.Tdw,"Q",Xe,'water')
        a=self.W4*Xe-self.Cl*self.Pressure_s+
        self.rho_g*self.Ad*self.DLdw()
        b=self.Vdr-self.Ad*self.Ldw
        dtdrho_g=a/b
        return dtdrho_g

    '''------------------------------ Down Comer region-------------------------'''

    def DTd(self):
        rho_d=PropsSI('D',"T",self.Tfi,'Q',0,'water')
```

```
        dtdTd=(self.Tdw-self.Td)*self.W1/
        (self.Ldw*self.Ad*rho_d)
        return dtdTd

    def integrator(self,function,argsforfunction:list,
    intitial_cond,time_step):
        l=len(argsforfunction)

        if l==0:
            return function()*time_step+intitial_cond
        elif l==1:
            arg1=argsforfunction[0]
            return function(arg1)*time_step+intitial_cond
        elif l==2:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            return function(arg1,arg2)*time_step+intitial_cond
        elif l==3:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            arg3=argsforfunction[2]
            return function(arg1,arg2,arg3)*time_step+
            intitial_cond
        else:
            raise   AttributeError("agrs in your differential
            function were not correct! Fix them")
```

---

primary_cooling_pump.py

```
import numpy as np
import scipy as sp
from CoolProp.CoolProp import PropsSI

class reactor_primary_coolant_pump():

    def __init__(self,T_po:float,pipe_diameter:float,
    piping_length:float,friction_factor:float,
    moment_of_innertia:float,pump_speed:float,
```

```
flow_rate:float,power_delivered_to_shaft:float,
const:list):

    self.A=np.pi*pipe_diameter**2/4
    self.k=friction_factor
    self.L=piping_length
    self.g=9.8
    self.water_density=PropsSI('D',"T",T_po,'Q',0,'water')
    self.const=const
    self.N=pump_speed
    self.Q=flow_rate
    self.Pd=power_delivered_to_shaft
    self.I=moment_of_innertia
    self.Head_update()
def Head_update(self):
    sum=0
    for i in range(len(self.const)):
        sum=sum+self.const[i]*self.Q**i
    self.H=sum

def DQ(self):

    Dq=self.A*self.g*(self.H-self.k*self.Q)/self.L

    return Dq

def DN(self,T_po:float):
    self.water_density=PropsSI('D',"T",T_po,'Q',0,'water')
    DNp=(self.Pd-self.g*self.water_density*self.Q*self.H)/
    (self.N*self.I*4*np.pi**2)
    return DNp

def integrator(self,function,argsforfunction:list,
intitial_cond,time_step):
    l=len(argsforfunction)

    if l==0:
        return function()*time_step+intitial_cond
    elif l==1:
```

```
            arg1=argsforfunction[0]
            return function(arg1)*time_step+intitial_cond
        elif l==2:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            return function(arg1,arg2)*time_step+intitial_cond
        elif l==3:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            arg3=argsforfunction[2]
            return function(arg1,arg2,arg3)*time_step+
            intitial_cond
        else:
            raise   AttributeError("agrs in your differential
            function were not correct! Fix them")
import numpy as np
from CoolProp.CoolProp import PropsSI
-----------------------------------------------------------------
                            CONDENSER_MODEL
 -----------------------------------------------------------------
 -----------------------------------------------------------------
 condenser_model_a.py
 -----------------------------------------------------------------

"""----------------------CONDNSER MODEL A-------------------------"""

class Condenser_Model_A():
    def __init__(self,exhust_flow_rate_lp:float,exhust_temp_lp:float,
    exhust_temp_pressure:float,
    Condenser_outlet_temp:float,Temp_cold_water_in:float,
    Temp_hot_water_out:float):

        self.Wi=exhust_flow_rate_lp
        self.Ti=exhust_temp_lp
        self.time_const=7
        self.Tout=Condenser_outlet_temp
        self.Rs=0.4615e3
        self.Hot_well_diameter=1
        self.volume=3
```

```
        self.UA=356.972e3
        self.Cp=4.2e3

        self.hout=PropsSI('H','T',self.Tout,'Q',0,'water')
        self.hf=PropsSI("H","T",self.Ti,'Q',0,'water')
        self.hi=PropsSI("H",'T',self.Ti,'P',exhust_temp_pressure,'water')
        self.hg=PropsSI("H",'T',self.Ti,'Q',1,'water')
        self.hfg=self.hg-self.hf
        self.W2=self.Wi*(self.hi-self.hf)/self.hfg
        self.W1=self.Wi*(1-((self.hi-self.hf)/self.hfg))
        self.T_coldwater=Temp_cold_water_in
        self.T_hotwater=Temp_hot_water_out

        self.deltaT=(self.T_hotwater-self.T_coldwater)/
        np.log((self.Ti-self.T_coldwater)/(self.Ti-self.T_hotwater))
        self.hin=PropsSI("H","T",self.Ti,'P',exhust_temp_pressure,'water')
        self.hcw=PropsSI("H","T",self.T_coldwater,'Q',0,'water')
        self.W3=(self.UA*self.deltaT)/
        (self.hin -self.hcw)

    def DW_condensate(self):
        dtdWsc=(self.W2-self.W3)/self.time_const
        return dtdWsc
    def DPs(self,differential_change_in_input_steam:float):

        DWs=differential_change_in_input_steam
        dtdPs=self.Rs*(DWs*self.Ti)/self.volume
        return dtdPs

    def Dhout(self):
        self.rho_w=PropsSI("D",'T',self.Tout,'Q',0,'water')
        area=np.pi*self.Hot_well_diameter**2/4
        self.Lw=self.DW_condensate()/(self.rho_w*area)

        dtdho=(self.W1+self.W3)*(self.hf-self.hout)/
        (self.Lw*area*self.rho_w)
        return dtdho

    def integrator(self,function,argsforfunction:list,intitial_cond,
```

```
    time_step):
        l=len(argsforfunction)

        if l==0 or argsforfunction== None:
            return function()*time_step+intitial_cond
        elif l==1:
            arg1=argsforfunction[0]
            return function(arg1)*time_step+intitial_cond
        elif l==2:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            return function(arg1,arg2)*time_step+intitial_cond
        elif l==3:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            arg3=argsforfunction[2]
            return function(arg1,arg2,arg3)*time_step+
            intitial_cond
        else:
            raise   AttributeError("agrs in your differential function
            were not correct! Fix them")

 ---------------------------------------------------------------
 condenser_model_b.py
 ---------------------------------------------------------------
"""------------------------------CONDNSER MODEL B------------------------------"""
class Condenser_Model_B():

    def __init__(self,steampressure:float,airpressure:float,hot_well_diametr:float):

        self.volume=3
        self.UA=356.972e3
        self.Cp=4.2e3
        self.Ps=steampressure
        self.Pa=airpressure
        self.Rs=0.4615e3
        self.Ra=.287e3

        """ steam mass balance variables"""
```

```
        self.W_turbine=4
        self.W_otherthanturbine=0
        self.W_condensate=4
        self.W_steamairout=0

        self.T_steamin=600
        self.T_steamout=600

        """ cold water """
        self.W_coldwater=107.881
        self.T_coldwater=273+60
        self.Ww=102310
        self.T_hotwater=273+80
        self.hotwelldensity=227
        self.Hs=PropsSI('H','T',self.T_steamin,'P',self.Ps,'water')
        self.Hcw=PropsSI('H','T',self.T_coldwater,'P',101325,'water')
        self.deltaT=(self.T_hotwater-self.T_coldwater)/
        np.log((self.T_steamin-self.T_coldwater)/
        (self.T_steamin-self.T_hotwater))


        """ air zone """
        self.W_vaccumbreakvalve=0
        self.W_air=0
        self.W_steamgas=0
        self.W_draincondenser=0

        """ hot well water"""
        self.Hot_wellarea=np.pi*hot_well_diametr**2/4
        self.W_hotwell=110
        self.W_bubblingoxygen=100

        self.Wc=(self.UA*self.deltaT)/(self.Hs-self.Hcw)

    def DWs(self):

        self.R=(self.Pa*self.Rs)/(self.Pa*self.Ra+self.Ps*self.Rs)
        self.Wss=self.W_air*(1-self.R)
        self.deltaT=(self.T_hotwater-self.T_coldwater)/
```

```
    np.log((self.T_steamin-self.T_coldwater)/
    (self.T_steamin-self.T_hotwater))
    self.Wc=(self.UA*self.deltaT)/(self.Hs-self.Hcw)
    dtdWsteam=self.W_turbine+self.W_otherthanturbine-
    self.Wc-self.Wss

    return dtdWsteam

def DPs(self):
    dtdPs=self.Rs*(self.DWs()*self.T_steamin)/self.volume
    return dtdPs

def Dhs(self,temp_otherthanturbine:float,
pressure_otherthanturbine:float):

    self.Hs=PropsSI('H','T',self.T_steamin,'P',self.Ps,'water')
    dtdWSHS=(self.W_turbine*self.Hs+self.W_otherthanturbine*
    PropsSI("H",'T',temp_otherthanturbine,'P',
    pressure_otherthanturbine,'water')-(self.Wc+self.Wss)*
    self.Hs)/self.WaterLevel

    return dtdWSHS

def DWa(self):
    dtdWa=self.W_vaccumbreakvalve+self.W_draincondenser+
    self.W_steamgas-self.W_air
    return dtdWa

def DPa(self):
    dtdpa=self.DWa()*self.Ra*self.T_steamin/self.volume
    return dtdpa

def TotalPressure(self):
    return self.Pa+self.Ps

def Lw(self):
    return self.Ww/(self.Hot_wellarea*self.hotwelldensity)

def DWw(self):
```

```
        dtdWhw=self.Wc+self.W_bubblingoxygen-self.Ww
        return dtdWhw
    def Dhw(self,Enthalpy_oxygen,pressure):
        dtdWwHw=self.W_coldwater*self.Hcw+
        self.W_bubblingoxygen*Enthalpy_oxygen-self.Ww*PropsSI('H',self.T_steamout,'P'
        return dtdWwHw
    def DTtube(self):
        dtdTh=(self.UA*self.deltaT-self.W_coldwater*
        self.Cp*(self.T_hotwater-self.T_coldwater))/
        (self.W_coldwater*self.Cp)
        return dtdTh

    def integrator(self,function,argsforfunction:list,
    intitial_cond,time_step):
        l=len(argsforfunction)

        if l==0 or argsforfunction== None:
            return function()*time_step+intitial_cond
        elif l==1:
            arg1=argsforfunction[0]
            return function(arg1)*time_step+intitial_cond
        elif l==2:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            return function(arg1,arg2)*time_step+intitial_cond
        elif l==3:
            arg1=argsforfunction[0]
            arg2=argsforfunction[1]
            arg3=argsforfunction[2]
            return function(arg1,arg2,arg3)*time_step+
            intitial_cond
        else:
            raise   AttributeError("agrs in your differential
            function were not correct! Fix them")
```

Turbine_system.py

```
import numpy as np
```

```
import scipy as sp
from CoolProp.CoolProp import PropsSI

class Turbine_system():
    def __init__(self,steam_flow_rate_in:float,steam_generator_outlet_temp:float,
    rho_sd:float,rho_hpex:float,T_hpex:float):
        self.Win=steam_flow_rate_in

        self.C1=0.2
        self.C2=1-self.C1
        self.Wsec=self.C2*self.Win
        self.Wmain=self.C1*self.Win

        """ Nozzle chest data """
        self.rho_c=38.8
        self.Pc=67.22e5
        self.hc=PropsSI("H",'D',self.rho_c,'P',self.Pc,'water')
        self.Tc=PropsSI('T','D',self.rho_c,'P',self.Pc,'water')
        self.Vc=212  #need to calculate
        self.k1=0.02 #need to calculate

        '''  Reheater data   '''
        self.Tsd=steam_generator_outlet_temp
        self.rho_sd=rho_sd
        self.rho_r=2.4
        self.hr=2920.2e3
        self.Pr=PropsSI('P','H',self.hr,'D',self.rho_r,'water')

        ''' High pressure turbine'''
        self.rho_hpex=rho_hpex
        self.T_hpex=T_hpex
        self.hpex=PropsSI("H","D",self.rho_hpex,'T',self.T_hpex,'water')
        self.Cbhp=0.2
        #this is at random. need no idea how to calculate it
        self.Whp=self.Kch*np.sqrt(self.Pc*self.rho_c-self.Pr*self.rho_hpex)
        self.Whpex=299.2
        self.Tau1=2

        ''' Low pressure turbine '''
```

```
        self.Wlp=269.92
        self.Clp=0.02
        #this is at random. need no idea how to calculate it
        self.Wlpex=237.15
        self.hlpex=2457e3
        self.tau2=10

        ''' Feed Water heater High pressure '''
        self.hfwh1=425.4
        ''' feed water heater low pressure  '''

    def Throttle_valve(self):
        self.Wsec=self.C2*self.Win
        self.Wmain=self.C1*self.Win

    def Drho_c(self):
        self.Pc=PropsSI("P",'H',self.hc,'P',self.rho_c,'water')
        self.Wh1=self.Kch*np.sqrt(self.Pc*self.rho_c-self.Pr*self.rho_hpex)
        dtdrho_c=(self.Wmain-self.Wh1)/self.Vc
        return dtdrho_c

    def W_mss(self):
        hf=PropsSI("H",'T',self.T_hpex,'Q',0,'water')
        hg=PropsSI("H",'T',self.T_hpex,'Q',0,'water')
        h_hpex=PropsSI("H",'T',self.T_hpex,'D',self.rho_hpex,'water')
        self.Wmss=self.Whpex*(h_hpex-hf)/(hg-hf)

    def W_msw(self):
        hf=PropsSI("H",'T',self.T_hpex,'Q',0,'water')
        hg=PropsSI("H",'T',self.T_hpex,'Q',0,'water')
        h_hpex=PropsSI("H",'T',self.T_hpex,'D',self.rho_hpex,'water')
        self.Wmsw=self.Whpex*(1-(h_hpex-hf)/(hg-hf))

    def Dh_c(self):
        hsd=PropsSI("H",'T',self.Tsd,'D',self.rho_sd,'water')
        self.Wh1=self.Kch*np.sqrt(self.Pc*self.rho_c-self.Pr*self.rho_hpex)
        a=(self.Wmain*hsd-self.whp1*self.hc)/(self.rho_c*self.Vc)
        b=(self.Pc*self.Drho_c())/self.rho_c**2
```

```
    dtdhc=(a+b)*(1/(1-self.k1))
    return dtdhc

def DW_hpex(self):
    dtdWhpex=(self.Wh1*(1-self.Cbhp)-self.Whpex)/self.Tau1
    return dtdWhpex

def DW_lpex(self):
    dtdWlpex=(self.Wlp*(1-self.Clp)-self.Wlpex)/self.Tau2
    return dtdWlpex


def Torque(self):
    h_hpex=PropsSI("H",'T',self.T_hpex,'D',self.rho_hpex,'water')
    Torque_hp=self.Wh1*(self.hc-h_hpex)
    Torque_lp=self.Wlp*(self.hr-self.hlpex)

    return Torque_hp+Torque_lp


def Dh_fwh1(self,hco:float,W_condenser_outlet:float):
    ''' Here needs some updates '''
    self.Wlp=self.Wr
    self.Wfw=W_condenser_outlet
    hh1=9232 # I don't know that value
    w1=self.Cbhp*self.Wh1+self.Wmss+self.Wr+self.Clp*self.Wlp
    H=(hco-self.hfwh1)
    Tau=60
    dtdh_fwh1=(w1*hh1/self.Wfw+H)/Tau
    return dtdh_fwh1

def integrator(self,function,argsforfunction:list,
intitial_cond,time_step):
    l=len(argsforfunction)

    if l==0 or argsforfunction== None:
        return function()*time_step+intitial_cond
    elif l==1:
        arg1=argsforfunction[0]
```

```
    return function(arg1)*time_step+intitial_cond
elif l==2:
    arg1=argsforfunction[0]
    arg2=argsforfunction[1]
    return function(arg1,arg2)*time_step+intitial_cond
elif l==3:
    arg1=argsforfunction[0]
    arg2=argsforfunction[1]
    arg3=argsforfunction[2]
    return function(arg1,arg2,arg3)*time_step+
    intitial_cond
else:
    raise   AttributeError("agrs in your differential
    function were not correct! Fix them")
```

```
------------------------------------------------------------
                DONE AZOG SOURCE CODE
------------------------------------------------------------
Copyright :
Ebny Walid Ahammed
Dept of Nuclear Engineering,
University of Dhaka
```