DUNE: Nuclear Reactor Point Kinetics Simulator

Comprehensive Technical Report and Code Analysis

An Educational Tool for University 1st Year Nuclear Engineering Students

Course Code: NE-3206
Group Members:

Hridoy Kabiraj (Roll: 14)

Aishik Chowdhury (Roll: 17)
Abrar Abdullah Diam (Roll: 56)
Department of Nuclear Engineering

University of Dhaka
GitHub Repository: https://github.com/Hridoy-Kabiraj/DUNE

February 6, 2026

Abstract

This comprehensive technical report presents a detailed analysis of the DUNE (Demonstrable Utility for Nuclear Education) project—a nuclear reactor point kinetics simulation system with an interactive GUI frontend and optional Arduino-driven 3D printed physical model integration. The project was developed as an educational tool to introduce University 1st year students to nuclear engineering concepts through hands-on simulation and visualization. This document provides an in-depth exploration of the theoretical foundations of reactor kinetics, detailed code walkthroughs with line-by-line annotations, architectural analysis, and implementation details. The simulation implements the six-group delayed neutron point kinetics equations coupled with thermal-hydraulic feedback, control rod dynamics, Xenon-135 and Samarium-149 fission product poisoning, and fuel burnup with isotope depletion tracking (U-235, U-238, Pu-239), offering real-time visualization of reactor behavior including power transients, temperature evolution, isotope concentrations, and safety systems such as SCRAM protection. The Arduino integration provides tangible physical feedback through servo-controlled control rods and LED power indicators, bridging the gap between abstract nuclear physics concepts and observable physical phenomena.

Contents

1 Introduction **7**
1.1 Project Context and Motivation . . . 7
1.2 Problem Statement . . . 7
1.3 Intended Audience and Use Cases . . . 7
1.4 Key Features and Capabilities . . . 8
1.5 Document Organization . . . 9

2 Repository Exploration **9**
2.1 Repository Structure Overview . . . 9
2.2 Core Python Modules . . . 10
2.2.1 reactorPhysics.py . . . 10
2.2.2 reactor.py . . . 10
2.2.3 DUNEReactor.py . . . 10
2.2.4 guiTemplate.py . . . 11
2.3 Arduino Integration . . . 11
2.3.1 reactorSketch.ino . . . 11
2.4 Package Configuration . . . 11
2.4.1 setup.py . . . 11
2.5 Documentation . . . 12
2.5.1 README.md . . . 12

3 Theoretical Background: Nuclear Reactor Point Kinetics **12**
3.1 Introduction to Reactor Kinetics . . . 12
3.2 Neutron Population Dynamics . . . 12
3.2.1 The Neutron Life Cycle . . . 12
3.2.2 The Point Kinetics Approximation . . . 13
3.3 Point Kinetics Equations with Delayed Neutrons . . . 13
3.3.1 Physical Interpretation . . . 13
3.4 Delayed Neutron Data for U-235 . . . 14
3.5 Reactivity and Its Components . . . 14
3.5.1 Reactivity Units . . . 14
3.5.2 Reactivity Components in DUNE . . . 14
3.6 Thermal-Hydraulic Coupling . . . 15
3.6.1 Fuel Temperature . . . 15
3.6.2 Coolant Temperature . . . 15
3.6.3 Power Calculation . . . 16
3.7 Control Rod Dynamics . . . 16
3.8 Reactor SCRAM . . . 16
3.9 Xenon-135 Poisoning Dynamics . . . 16
3.9.1 Production and Removal Pathways . . . 17
3.9.2 Iodine-135 Balance Equation . . . 17
3.9.3 Xenon-135 Balance Equation . . . 17
3.9.4 Xenon Reactivity Contribution . . . 17
3.9.5 Total Reactivity with Xenon . . . 18
3.9.6 Physical Phenomena Modeled . . . 18
3.10 Samarium-149 Poisoning Dynamics . . . 18
3.10.1 Production and Removal Pathways . . . 18

3.10.2 Neodymium-149 Balance Equation . . . 18
3.10.3 Promethium-149 Balance Equation . . . 19
3.10.4 Samarium-149 Balance Equation . . . 19
3.10.5 Samarium Reactivity Contribution . . . 19
3.10.6 Total Reactivity with Both Poisons . . . 19
3.10.7 Physical Phenomena Modeled . . . 19
3.10.8 Comparison: Xenon-135 vs Samarium-149 . . . 19
3.11 Burnup and Isotope Depletion . . . 20
3.11.1 Burnup Equation . . . 20
3.11.2 U-235 Depletion . . . 20
3.11.3 U-238 Transmutation . . . 20
3.11.4 Pu-239 Production and Consumption . . . 21
3.11.5 Fission Product Accumulation . . . 21
3.11.6 Burnup Reactivity Contribution . . . 21
3.11.7 Initial Isotope Concentrations . . . 21
3.11.8 Excess Reactivity . . . 22
3.12 Extended State Vector . . . 22
3.13 Complete State Vector . . . 22

4 Detailed Code Walkthrough: reactorPhysics.py 22
4.1 Module Header and Physical Constants . . . 23
4.2 Reactivity Feedback Coefficient . . . 24
4.3 Neutron Population Dynamics . . . 25
4.4 Precursor Population Dynamics . . . 25
4.5 Thermal Power Calculation . . . 26
4.6 Fuel Temperature Dynamics . . . 27
4.7 Coolant Temperature Dynamics . . . 28
4.8 Control Rod Worth Functions . . . 28
4.9 Total Reactivity Calculation . . . 29
4.10 Complete Reactor System ODEs . . . 30

5 Reactor Control System: reactor.py 31
5.1 DUNEReactor Class Structure . . . 31
5.2 Time Stepping and Integration . . . 32
5.3 Pre-Step Control Logic . . . 33
5.4 SCRAM Protection System . . . 33
5.5 Manual Rod Control . . . 34
5.6 PID Power Control . . . 35
5.7 Coolant Flow Control . . . 36
5.7.1 Dynamic Flow Rate Adjustment . . . 36
5.7.2 Manual Coolant Control Mode . . . 37
5.7.3 Gradual Flow Rate Ramping . . . 37
5.7.4 Prompt Jump Mode . . . 37
5.8 Testing and Validation . . . 38

6 GUI Frontend: duneReactor.py **39**
6.1 GUI Architecture Overview . . . 39
6.2 Main Application Class . . . 39
6.3 Plot Initialization and Configuration . . . 40
6.4 Real-Time Plotting . . . 41
6.5 Event Handlers: User Interaction . . . 42
6.6 Monitor Updates . . . 43
6.7 CSV Data Logging System . . . 44
6.7.1 CSV Initialization . . . 44
6.7.2 Data Logging at Fixed Intervals . . . 45
6.7.3 CSV File Closure on Exit . . . 46
6.7.4 CSV Data Structure . . . 46
6.8 Arduino Serial Communication . . . 47
6.9 Serial Port Auto-Detection . . . 48
6.10 Application Entry Point . . . 49

7 Arduino Hardware Integration **49**
7.1 Arduino Sketch Overview . . . 49
7.1.1 Detailed Circuit Implementation with L298N Motor Driver . . . 50
7.2 Serial Command Processing . . . 51
7.2.1 Python-Arduino Communication for Coolant Control . . . 52
7.2.2 Integer Parsing from Serial . . . 53
7.3 Physical Model Integration . . . 53

8 Mathematical Formulation and Numerical Methods **54**
8.1 System of ODEs . . . 54
8.2 Reactivity Functions . . . 54
8.2.1 Control Rod Worth . . . 54
8.2.2 Temperature Reactivity . . . 55
8.3 Numerical Integration Method . . . 55
8.3.1 LSODA Algorithm . . . 55
8.3.2 Why LSODA for Reactor Kinetics? . . . 55
8.4 Time Stepping Strategy . . . 55
8.5 Stability Analysis: Linearized System . . . 56
8.6 Power-Temperature Coupling . . . 56
8.6.1 Power Increase Scenario . . . 56

9 Usage Examples and Operational Scenarios **56**
9.1 Installation and Setup . . . 56
9.1.1 Prerequisites . . . 56
9.1.2 Launching the Application . . . 57
9.2 Operational Scenarios . . . 57
9.2.1 Scenario 1: Reactor Startup from Cold Shutdown . . . 57
9.2.2 Scenario 2: Automatic Power Control . . . 57
9.2.3 Scenario 3: Reactivity-Initiated Accident (RIA) . . . 58
9.2.4 Scenario 4: Loss of Coolant Flow . . . 58
9.2.5 Scenario 5: Plot Zoom and Time History Analysis . . . 59
9.2.6 Scenario 6: Prompt Jump Mode Demonstration . . . 59
9.3 Arduino Hardware Demonstrations . . . 60

9.3.1 Visual Power Feedback . . . 60
9.3.2 Control Rod Motion . . . 60
9.3.3 SCRAM Indication . . . 60
9.3.4 Auditory Feedback from Coolant Pump . . . 60
9.4 Suggested Classroom Activities . . . 60
9.4.1 Activity 1: Find the Critical Rod Position . . . 60
9.4.2 Activity 2: Control System Challenge . . . 61
9.4.3 Activity 3: SCRAM Limit Investigation . . . 61
9.4.4 Activity 4: Transient Analysis . . . 61

10 Conclusions and Future Enhancements **61**
10.1 Project Summary . . . 61
10.2 Key Accomplishments . . . 61
10.2.1 Technical Achievements . . . 61
10.2.2 Educational Impact . . . 62
10.3 Limitations of Current Implementation . . . 62
10.3.1 Physics Simplifications . . . 62
10.3.2 Control System Limitations . . . 63
10.4 Future Enhancement Opportunities . . . 64
10.4.1 Physics Improvements . . . 64
10.4.2 Software Enhancements . . . 64
10.4.3 Hardware Expansions . . . 65
10.4.4 Educational Curriculum Development . . . 65
10.5 Broader Impact . . . 66
10.5.1 STEM Education . . . 66
10.5.2 Cyber-Physical Systems in Engineering Education . . . 66
10.5.3 Public Understanding of Nuclear Energy . . . 66
10.5.4 Open-Source Community . . . 66
10.6 Project Heritage and Attribution . . . 66
10.7 Final Remarks . . . 67

Acknowledgements **67**

A Reactor Physics Equations Reference **69**
A.1 Point Kinetics Equations (Standard Form) . . . 69
A.2 Reactor Period . . . 69
A.3 Prompt Jump Approximation . . . 69

B Thermodynamic Properties **69**
B.1 UO_2 Fuel Properties . . . 69
B.2 Water/Steam Properties (at 15 MPa) . . . 69

1 Introduction

1.1 Project Context and Motivation

Nuclear energy represents one of the most powerful and complex technologies developed by humanity, yet public understanding of nuclear reactor physics remains limited and often clouded by misconceptions. The DUNE (Demonstrable Utility for Nuclear Education) project addresses this knowledge gap by providing an accessible, interactive educational tool that demystifies nuclear reactor operation through simulation and visualization.

The project emerged from the recognition that effective STEM education, particularly in nuclear engineering, requires hands-on experiences that bridge theoretical knowledge with observable phenomena. Traditional lecture-based approaches often fail to convey the dynamic, interconnected nature of reactor systems. DUNE solves this pedagogical challenge by implementing a scientifically accurate point kinetics reactor model with real-time graphical feedback and optional physical integration through Arduino-controlled 3D printed models.

1.2 Problem Statement

The core challenges addressed by this project include:

1. **Accessibility**: Making nuclear reactor physics accessible to University 1st year students with fundamental mathematical background.
2. **Engagement**: Creating an interactive experience that maintains student interest while conveying complex technical concepts.
3. **Safety Education**: Demonstrating reactor safety systems (particularly SCRAM protection) in a risk-free simulated environment.
4. **Physical Intuition**: Connecting abstract equations to tangible physical phenomena through Arduino-driven hardware feedback.
5. **Real-time Dynamics**: Illustrating the time-dependent behavior of reactor systems, including transients, control strategies, and feedback mechanisms.

1.3 Intended Audience and Use Cases

DUNE is designed for multiple audiences:

- **University 1st Year Students**: Primary target audience for educational demonstrations and interactive learning experiences in nuclear engineering courses.
- **Educators**: Science teachers seeking engaging tools for nuclear energy education and STEM outreach.
- **University Students**: Undergraduate nuclear engineering students learning reactor kinetics fundamentals.
- **Public Outreach**: Science fairs, open house events, and community education programs about nuclear technology.
- **Researchers**: As a platform for developing and testing educational methodologies in nuclear engineering education.

1.4 Key Features and Capabilities

The DUNE system provides:

- Real-time solution of six-group delayed neutron point kinetics equations
- **Xenon-135 and Samarium-149 poisoning dynamics** with complete decay chain modeling
- **Burnup and isotope depletion tracking**: U-235, U-238, Pu-239, and fission product concentrations
- Coupled thermal-hydraulic modeling with fuel and coolant temperature tracking
- Interactive **full-screen GUI** with control rod manipulation and power level control
- Automatic reactor protection system (SCRAM) with configurable safety limits
- Dual control modes: manual control rod positioning and automatic power control (PID)
- **Prompt Jump Mode**: Instantly inserts \$0.003 reactivity for demonstrating prompt neutron response
- **Eight-panel live plotting**: reactor power, reactivity (\$), temperatures, fission product poisons, burnup, and isotope concentrations
- **Real-time monitoring displays**: Reactivity (\$), Xenon-135, Samarium-149, Burnup, U-235, U-238, Pu-239 concentrations
- **Dynamic reactivity display**: Real-time reactivity value in dollars, showing zero at criticality
- Optional Arduino integration for physical model control (servo motors, LED feedback)
- Temperature-dependent reactivity feedback modeling
- **Enhanced coolant system**: Multi-poison aware reactivity, burnout effects on poisoning
- Dynamic coolant flow rate adjustment based on reactor power (200–1200 kg/s)
- Manual coolant flow control mode with user-defined flow rates
- Automatic CSV data logging including burnup, isotope concentrations, Xenon, Samarium and reactivity traces
- Independent pump speed control via Arduino case 'c' command
- Real-time reactivity decomposition: temperature, rod position, Xenon, Samarium, and burnup contributions
- **Scrollable value panel**: Organized sections for Power/Reactivity, Temperature, FP Poisons, and Burnup/Isotopes
- **Improved GUI layout**: Rod position controls consolidated in left panel for better ergonomics

1.5 Document Organization

This report is structured as follows:

- **Section 2**: Repository structure and file organization
- **Section 3**: Theoretical background on nuclear reactor point kinetics
- **Section 4**: Detailed code analysis of reactor physics module
- **Section 5**: Reactor control system implementation
- **Section 6**: GUI frontend architecture and implementation
- **Section 7**: Arduino integration for physical model control
- **Section 8**: Mathematical formulations and numerical methods
- **Section 9**: Usage examples and operational scenarios
- **Section 10**: Conclusions and future enhancements

2 Repository Exploration

2.1 Repository Structure Overview

The DUNE project is organized into a logical directory structure that separates concerns between physics modeling, user interface, hardware integration, and deployment:

```
DUNE/
 README.md                    # Project documentation
 setup.py                     # Python package installation script
 reactor.py                   # Core reactor control class
 reactorPhysics.py           # Point kinetics equations and physics
 DUNEReactor.py              # Main GUI application
 guiTemplate.py              # wxPython GUI layout (auto-generated)
 generate_report.py          # Utility script
 __pycache__/                # Python bytecode cache
 arduino/
    reactorSketch/
        reactorSketch.ino   # Arduino firmware for hardware control
 build/                       # Build artifacts
    bdist.linux-x86_64/
    lib/                     # Compiled library files
 doc/
    readme.tex              # LaTeX documentation source
 DUNE.egg-info/         # Python package metadata
     dependency_links.txt
     entry_points.txt
     PKG-INFO
     requires.txt
     SOURCES.txt
     top_level.txt
```

2.2 Core Python Modules

2.2.1 reactorPhysics.py

This module contains the fundamental nuclear reactor physics implementation:

- **Purpose**: Implements point kinetics equations with six delayed neutron groups
- **Key Components**:
  - Delayed neutron group parameters (β_i, λ_i)
  - Neutron population dynamics (`dndt`)
  - Precursor concentration evolution (`dCdt`)
  - Thermal power calculation (`qFuel`)
  - Temperature derivatives for fuel and coolant (`dTfdt`, `dTcdt`)
  - Control rod reactivity worth curves (`diffRodWorth`, `intRodWorth`)
  - Total reactivity calculation (`rho`)
  - Complete reactor system ODEs (`reactorSystem`)
- **Mathematical Framework**: Uses numpy for array operations and defines physical constants based on U-235 fission data

2.2.2 reactor.py

The reactor control and state management module:

- **Purpose**: Provides high-level interface for reactor simulation
- **Key Class**: `DUNEReactor`
- **Responsibilities**:
  - State vector management (neutrons, precursors, temperatures, rod position)
  - Time stepping using scipy ODE integrators
  - Control logic (manual rod control vs. automatic power control)
  - PID controller implementation for power regulation
  - SCRAM protection system
  - Data storage for plotting and analysis

2.2.3 DUNEReactor.py

The main application with GUI integration:

- **Purpose**: User interface and real-time visualization
- **Framework**: wxPython for GUI, matplotlib for plotting
- **Key Features**:

  - Real-time plot updates (power, fuel temperature, coolant temperature)
  - User input handling (sliders, text boxes, buttons)
  - Timer-based event loop for continuous simulation
  - Arduino serial communication for 3D printed physical model
  - Interactive control element binding

2.2.4 guiTemplate.py

Auto-generated GUI layout code:

- **Purpose**: Defines GUI widget layout and structure
- **Generation Tool**: wxFormBuilder
- **Components**: Panels, sliders, text controls, buttons, gauges
- **Note**: This file should not be manually edited; regenerate from wxFormBuilder project

2.3 Arduino Integration

2.3.1 reactorSketch.ino

Arduino firmware for physical model control:

- **Purpose**: Provides hardware feedback for reactor simulation
- **Hardware Interface**:
  - Servo motor (pin 9): Control rod position indication in 3D printed model
  - RGB LED Blue (pin 6): Reactor power level visualization
  - RGB LED Red (pin 11): SCRAM condition indicator
  - PWM Motor (pin 3): Coolant pump speed control
- **Communication Protocol**: Serial commands at 9600 baud
  - 'p' + number: Set power LED brightness (0-255)
  - 'r' + number: Set control rod servo position (0-180)
  - 's' + number: Set SCRAM LED state (0 or 1)

2.4 Package Configuration

2.4.1 setup.py

Python package configuration using setuptools:

- **Package Name**: DUNE
- **Entry Point**: `DUNE` command launches GUI application

- **Dependencies**:
  - numpy $\geq$ 1.20: Numerical computations
  - scipy $\geq$ 1.6: ODE integration
  - matplotlib $\geq$ 3.3: Plotting
  - pyserial $\geq$ 3.0: Arduino communication
  - wxPython $\geq$ 4.1: GUI framework
- **Python Version**: Requires Python 3.6 or newer

2.5 Documentation

2.5.1 README.md

Project overview and usage instructions:

- Installation procedures
- Usage instructions
- Control mode descriptions
- Arduino connection guidance
- Author information and licensing

3 Theoretical Background: Nuclear Reactor Point Kinetics

3.1 Introduction to Reactor Kinetics

Reactor kinetics describes the time-dependent behavior of neutron population in a nuclear reactor. Unlike static criticality analysis, kinetics accounts for the dynamic response of the neutron flux to changes in reactivity, which is essential for understanding reactor control, transient behavior, and safety systems.

3.2 Neutron Population Dynamics

3.2.1 The Neutron Life Cycle

In a nuclear reactor, neutrons are born from two sources:

1. **Prompt Neutrons**: Released immediately ($\sim 10^{-14}$ seconds) during nuclear fission
2. **Delayed Neutrons**: Released from the decay of fission products (precursors) with characteristic half-lives ranging from fractions of a second to nearly a minute

Although delayed neutrons constitute only about 0.65% of all neutrons from U-235 fission, they are crucial for reactor control. Without delayed neutrons, the reactor would respond to reactivity changes on prompt neutron timescales ($\sim 10^{-5}$ seconds), making control impossible.

3.2.2 The Point Kinetics Approximation

The point kinetics model assumes that the neutron flux shape remains constant over time, and only its amplitude varies. This simplification reduces the complex space-time neutron diffusion equations to a set of ordinary differential equations (ODEs) describing the total neutron population.

3.3 Point Kinetics Equations with Delayed Neutrons

The fundamental point kinetics equations with delayed neutron groups are:

$$\frac{dn(t)}{dt} = \frac{\rho(t) - \beta}{\Lambda} n(t) + \sum_{i=1}^{6} \lambda_i C_i(t) \tag{1}$$

$$\frac{dC_i(t)}{dt} = \frac{\beta_i}{\Lambda} n(t) - \lambda_i C_i(t) \quad \text{for } i = 1, 2, \ldots, 6 \tag{2}$$

where:

- $n(t)$ = neutron density [neutrons/cm^3]
- $C_i(t)$ = concentration of i-th delayed neutron precursor group [nuclei/cm^3]
- $\rho(t)$ = reactivity [dimensionless, often expressed in dollars or pcm]
- β = total delayed neutron fraction = $\sum_{i=1}^{6} \beta_i \approx 0.0065$ for U-235
- β_i = delayed neutron fraction for group i
- λ_i = decay constant for group i [s^{-1}]
- Λ = prompt neutron generation time [s] $\approx 10^{-5}$ s for thermal reactors

3.3.1 Physical Interpretation

Equation (1) describes the rate of change of neutron population:

- **First term** $\frac{\rho(t)-\beta}{\Lambda} n(t)$: Contribution from prompt neutrons. The factor $(\rho - \beta)$ represents the excess reactivity above delayed neutron fraction.
- **Second term** $\sum_{i=1}^{6} \lambda_i C_i(t)$: Contribution from delayed neutrons released by precursor decay.

Equation (2) describes precursor population evolution:

- **Production term** $\frac{\beta_i}{\Lambda} n(t)$: Precursors created from fission events
- **Decay term** $-\lambda_i C_i(t)$: Precursors lost to radioactive decay (releasing delayed neutrons)

3.4 Delayed Neutron Data for U-235

The DUNE simulation uses six-group delayed neutron parameters for U-235 thermal fission:

Table 1: Six-Group Delayed Neutron Parameters for U-235

Group i	β_i	λ_i [s^{-1}]
1	0.000215	0.0124
2	0.001424	0.0305
3	0.001274	0.111
4	0.002568	0.301
5	0.000748	1.14
6	0.000273	3.01
Total	$\beta = 0.0065$	-

3.5 Reactivity and Its Components

Reactivity is a dimensionless measure of how far the reactor is from critical:

$$\rho = \frac{k_{eff} - 1}{k_{eff}} \approx k_{eff} - 1 \tag{3}$$

where k_{eff} is the effective multiplication factor.

3.5.1 Reactivity Units

- **Absolute**: $\Delta k/k$ (dimensionless)
- **Percent milli-rho (pcm)**: $10^5 \times \rho$
- **Dollars (\$)**: ρ/β (normalized to delayed neutron fraction)

A reactivity of +\$1 (one dollar) means $\rho = \beta$, bringing the reactor to prompt critical.

3.5.2 Reactivity Components in DUNE

The total reactivity in DUNE includes:

$$\rho(t) = \rho_{rod}(h) + \rho_{temp}(T_{fuel}) \tag{4}$$

1. **Control Rod Reactivity** $\rho_{rod}(h)$:

   The integral rod worth as a function of rod height h (0 = fully inserted, 100 = fully withdrawn):

   $$\rho_{rod}(h) = \int_0^h R(h')\, dh' \tag{5}$$

   where the differential rod worth is:

$$R(h) = k \sin\left(\frac{\pi h}{100}\right) \tag{6}$$

This sinusoidal shape reflects the importance-weighted control rod worth, which is maximum near the core midplane.

2. **Temperature Reactivity Feedback** ρ_{temp}:

$$\rho_{temp}(T_{fuel}) = \alpha_T (T_{fuel} - T_{in}) \tag{7}$$

where α_T is the temperature coefficient of reactivity [pcm/K]. For most reactors, $\alpha_T < 0$ (negative feedback), providing inherent safety.

3.6 Thermal-Hydraulic Coupling

The reactor power generates heat in the fuel, which is removed by coolant. The thermal dynamics are modeled by:

3.6.1 Fuel Temperature

$$\frac{dT_{fuel}}{dt} = \frac{Q(t) - hA_c(T_{fuel} - T_{coolant})}{m_{fuel} C_{p,fuel}} \tag{8}$$

where:

- $Q(t)$ = thermal power from fission [W]
- h = heat transfer coefficient [W/cm^2·K]
- A_c = fuel-coolant contact area [cm^2]
- m_{fuel} = fuel mass [g]
- $C_{p,fuel}$ = fuel specific heat capacity [J/g·K]

3.6.2 Coolant Temperature

$$\frac{dT_{coolant}}{dt} = \frac{hA_c(T_{fuel} - T_{coolant}) + C_{p,H_2O}(T_{in} - T_{coolant})\dot{m}_c}{m_{coolant} C_{p,H_2O}} \tag{9}$$

where:

- $\dot{m}_c$ = coolant mass flow rate [g/s]
- T_{in} = coolant inlet temperature [K]

3.6.3 Power Calculation

The thermal power is related to neutron density by:

$$Q(t) = V_r \cdot V_{f,fuel} \cdot n(t) \cdot v \cdot \Sigma_f \cdot E_f \tag{10}$$

where:

- V_r = reactor volume [cm^3]
- $V_{f,fuel}$ = volume fraction of fuel
- v = neutron velocity [cm/s]
- Σ_f = macroscopic fission cross section [cm^{-1}]
- E_f = energy released per fission [J] $\approx 3.2 \times 10^{-11}$ J

3.7 Control Rod Dynamics

Control rods absorb neutrons, reducing reactivity. Their position $h(t)$ changes at rate $\dot{h}$:

$$\frac{dh}{dt} = \dot{h}(t) \tag{11}$$

The rod movement rate is limited to realistic values (e.g., 0.5%/s) to simulate mechanical constraints.

3.8 Reactor SCRAM

SCRAM (Safety Control Rod Ax Man) is an emergency shutdown mechanism. In DUNE, SCRAM is triggered when:

- Fuel temperature $T_{fuel} > 1700$ K
- Coolant temperature $T_{coolant} > 700$ K
- Manual SCRAM button activation

During SCRAM, control rods are immediately inserted ($h = 0$), introducing large negative reactivity and shutting down the reactor.

3.9 Xenon-135 Poisoning Dynamics

Xenon-135 is a fission product that acts as a powerful neutron absorber (poison). Its dynamics significantly affect reactor operation, especially during power changes and shutdowns.

3.9.1 Production and Removal Pathways

Xenon-135 has four main pathways:

1. **Direct production from fission**: A small fraction ($\gamma_X \approx 0.003$) of fissions produce Xe-135 directly
2. **Production from Iodine-135 decay**: Most Xe-135 comes from I-135 decay ($t_{1/2} \approx 6.6$ hr)
3. **Radioactive decay**: Xe-135 decays to Cs-135 ($t_{1/2} \approx 9.2$ hr)
4. **Neutron absorption (burnout)**: The large absorption cross-section ($\sigma_{a,X} \approx 2.6 \times 10^6$ barns) causes significant neutron capture

3.9.2 Iodine-135 Balance Equation

$$\frac{dI}{dt} = \gamma_I \Sigma_f \phi - \lambda_I I \tag{12}$$

where:

- $I(t)$ = Iodine-135 concentration [atoms/cm^3]
- $\gamma_I \approx 0.061$ = I-135 fission yield
- $\phi = \eta \cdot n$ = neutron flux [neutrons/cm^2·s]
- $\lambda_I = 2.87 \times 10^{-5}$ s^{-1} = I-135 decay constant

3.9.3 Xenon-135 Balance Equation

$$\frac{dX}{dt} = \gamma_X \Sigma_f \phi + \lambda_I I - \lambda_X X - \sigma_{a,X} \phi X \tag{13}$$

where:

- $X(t)$ = Xenon-135 concentration [atoms/cm^3]
- $\gamma_X \approx 0.003$ = Xe-135 direct fission yield
- $\lambda_X = 2.09 \times 10^{-5}$ s^{-1} = Xe-135 decay constant
- $\sigma_{a,X} = 2.6 \times 10^6 \times 10^{-24}$ cm^2 = Xe-135 absorption cross-section

3.9.4 Xenon Reactivity Contribution

Xenon introduces negative reactivity:

$$\rho_{Xe}(t) = -\frac{\sigma_{a,X} \eta X(t)}{\nu \Sigma_f \beta} \tag{14}$$

where $\nu \approx 2.43$ is the average number of neutrons per fission for U-235.

3.9.5 Total Reactivity with Xenon

The complete reactivity expression becomes:

$$\rho_{total}(t) = \rho_{rod}(h) + \rho_{temp}(T_{fuel}) + \rho_{Xe}(X) \tag{15}$$

3.9.6 Physical Phenomena Modeled

- **Xenon Buildup**: After startup, Xe-135 concentration increases over 20-40 hours to equilibrium
- **Equilibrium Xenon**: Production balances decay and burnout at steady power
- **Xenon Transients**: Power increases boost burnout, temporarily reducing Xe poisoning
- **Xenon Pit**: During shutdown, I-135 continues decaying to Xe-135 without burnout, causing a poisoning peak at ∼10-12 hours that may prevent reactor restart
- **Load Following**: Xenon dynamics complicate power changes, requiring anticipatory control strategies

3.10 Samarium-149 Poisoning Dynamics

Samarium-149 is another critical fission product poison with characteristics distinct from Xenon-135. It has a very large absorption cross-section and builds up through a multi-step decay chain.

3.10.1 Production and Removal Pathways

Samarium-149 formation involves a three-isotope chain:

1. **Neodymium-149**: Direct fission product ($\gamma_{Nd} \approx 0.011$, $t_{1/2} \approx 1.73$ hr)
2. **Promethium-149**: Intermediate isotope ($t_{1/2} \approx 53.1$ hr, moderate absorption)
3. **Samarium-149**: Final stable product (essentially no radioactive decay, very high absorption)

3.10.2 Neodymium-149 Balance Equation

$$\frac{dNd}{dt} = \gamma_{Nd}\Sigma_f\phi - \lambda_{Nd}Nd \tag{16}$$

where:

- $Nd(t)$ = Neodymium-149 concentration [atoms/cm^3]
- $\gamma_{Nd} \approx 0.011$ = Nd-149 fission yield
- $\lambda_{Nd} = 9.67 \times 10^{-5}$ s^{-1} = Nd-149 decay constant

3.10.3 Promethium-149 Balance Equation

$$\frac{dPm}{dt} = \lambda_{Nd} Nd - \lambda_{Pm} Pm - \sigma_{a,Pm} \phi Pm \tag{17}$$

where:

- $Pm(t)$ = Promethium-149 concentration [atoms/cm^3]
- $\lambda_{Pm} = 1.46 \times 10^{-6}$ s^{-1} = Pm-149 decay constant
- $\sigma_{a,Pm} = 1,400 \times 10^{-24}$ cm^2 = Pm-149 absorption cross-section

3.10.4 Samarium-149 Balance Equation

$$\frac{dSm}{dt} = \lambda_{Pm} Pm - \sigma_{a,Sm} \phi Sm \tag{18}$$

where:

- $Sm(t)$ = Samarium-149 concentration [atoms/cm^3]
- $\sigma_{a,Sm} = 40,800 \times 10^{-24}$ cm^2 = Sm-149 absorption cross-section (very large!)
- No decay term: Sm-149 half-life $\sim 2 \times 10^{15}$ years (essentially stable)

3.10.5 Samarium Reactivity Contribution

$$\rho_{Sm}(t) = -\frac{\sigma_{a,Sm} \eta Sm(t)}{\nu \Sigma_f \beta} \tag{19}$$

3.10.6 Total Reactivity with Both Poisons

$$\rho_{total}(t) = \rho_{rod}(h) + \rho_{temp}(T_{fuel}) + \rho_{Xe}(X) + \rho_{Sm}(Sm) \tag{20}$$

3.10.7 Physical Phenomena Modeled

- **Slow Buildup**: Sm-149 reaches equilibrium over days (vs hours for Xe-135)
- **Equilibrium Poisoning**: At steady power, typically contributes -0.5 to -1.0% $\Delta k/k$
- **Permanent Poison**: No radioactive decay; removed only by neutron absorption
- **Burnup Dependence**: Accumulates over core lifetime, unlike xenon which equilibrates quickly
- **Shutdown Behavior**: Unlike xenon, shows minimal transient peak after shutdown
- **Power Coefficient**: Higher power increases burnout rate, reducing Sm poisoning

3.10.8 Comparison: Xenon-135 vs Samarium-149

[h]

Property	Xe-135	Sm-149
Absorption cross-section	2.6 million barns	40,800 barns
Time to equilibrium	20-40 hours	Several days
Decay half-life	9.2 hours	Stable
Shutdown transient	Large peak (∼12 hr)	Minimal
Equilibrium worth	-2 to -3% $\Delta k/k$	-0.5 to -1% $\Delta k/k$

Table 2: Comparison of Xenon-135 and Samarium-149 poisoning characteristics

3.11 Burnup and Isotope Depletion

Fuel burnup and isotope depletion are essential for modeling long-term reactor behavior. The simulation tracks the evolution of key fuel isotopes as the reactor operates.

3.11.1 Burnup Equation

Burnup measures the total energy extracted per unit mass of fuel:

$$\frac{dB}{dt} = \frac{P(t)}{M_{HM} \times 86400} \tag{21}$$

where:

- B = Burnup [MWd/kgU]
- $P(t)$ = Reactor thermal power [MW]
- M_{HM} = Heavy metal mass = 11,100 kg (typical PWR core)
- 86400 = Seconds per day conversion factor

3.11.2 U-235 Depletion

U-235 is consumed primarily through fission:

$$\frac{dN_{235}}{dt} = -\sigma_{f,235} \cdot \phi \cdot N_{235} \tag{22}$$

where $\sigma_{f,235} = 585$ barns is the thermal fission cross-section for U-235.

3.11.3 U-238 Transmutation

U-238 undergoes neutron capture to eventually form Pu-239:

$$\frac{dN_{238}}{dt} = -\sigma_{c,238} \cdot \phi \cdot N_{238} \tag{23}$$

where $\sigma_{c,238} = 2.68$ barns is the thermal capture cross-section for U-238.

3.11.4 Pu-239 Production and Consumption

Pu-239 is produced from U-238 capture and consumed by fission:

$$\frac{dN_{239Pu}}{dt} = \sigma_{c,238} \cdot \phi \cdot N_{238} - \sigma_{f,239} \cdot \phi \cdot N_{239Pu} \tag{24}$$

where $\sigma_{f,239} = 750$ barns is the thermal fission cross-section for Pu-239.

3.11.5 Fission Product Accumulation

Fission products accumulate from both U-235 and Pu-239 fission:

$$\frac{dN_{FP}}{dt} = 2 \cdot (\sigma_{f,235} \cdot \phi \cdot N_{235} + \sigma_{f,239} \cdot \phi \cdot N_{239Pu}) \tag{25}$$

The factor of 2 accounts for the average yield of two fission products per fission event.

3.11.6 Burnup Reactivity Contribution

Burnup affects reactivity through three mechanisms:

- **U-235 depletion**: Loss of fissile material reduces reactivity
- **Pu-239 buildup**: Production of new fissile material adds reactivity
- **Fission product accumulation**: Parasitic absorption reduces reactivity

The net burnup reactivity is:

$$\rho_{burnup} = k_{235}(N_{235} - N_{235,0}) + k_{239}(N_{239Pu} - N_{239Pu,0}) - k_{FP} \cdot N_{FP} \tag{26}$$

where the reactivity coefficients are scaled for demonstration timescales.

3.11.7 Initial Isotope Concentrations

For a typical 4% enriched UO_2 PWR fuel:

- $N_{235,0} = 9.84 \times 10^{20}$ atoms/cm^3 (4% enrichment)
- $N_{238,0} = 2.21 \times 10^{22}$ atoms/cm^3 (96% of uranium)
- $N_{239Pu,0} = 0$ atoms/cm^3 (fresh fuel)
- $N_{FP,0} = 0$ atoms/cm^3 (fresh fuel)
- $B_0 = 0$ MWd/kgU (fresh fuel)

3.11.8 Excess Reactivity

Fresh reactor cores are designed with excess reactivity to compensate for:

- Fuel burnup over the cycle
- Fission product poisoning buildup
- Temperature defect at operating conditions

The simulation includes an initial excess reactivity of \$0.05 that must be compensated by control rod insertion. With a total control rod worth of \$0.2 following a sinusoidal differential worth curve, this means:

- At rod position = 0% (fully inserted): $\rho = -\$0.05$ (subcritical)
- At rod position $\approx$ 33%: $\rho = \$0.00$ (critical)
- At rod position = 100% (fully withdrawn): $\rho = +\$0.15$ (supercritical)

3.12 Extended State Vector

With Xenon-135, Samarium-149 poisoning, and burnup/isotope depletion implemented, the complete reactor system is now described by a 20-dimensional state vector:

$$\mathbf{S}(t) = [n, C_1, \ldots, C_6, T_{fuel}, T_{coolant}, h, I, X, Nd, Pm, Sm, N_{235}, N_{238}, N_{239Pu}, N_{FP}, B]^T \tag{27}$$

where the additional state variables are:

- $N_{235}(t)$ = U-235 concentration [atoms/cm^3]
- $N_{238}(t)$ = U-238 concentration [atoms/cm^3]
- $N_{239Pu}(t)$ = Pu-239 concentration [atoms/cm^3]
- $N_{FP}(t)$ = Fission product concentration [atoms/cm^3]
- $B(t)$ = Burnup [MWd/kgU]

The system is solved as a coupled set of ODEs using numerical integration.

3.13 Complete State Vector

Note: With the implementation of Xenon-135, Samarium-149 poisoning, and fuel burnup tracking, the state vector has been expanded to 20 dimensions, tracking the complete decay chains of both major fission product poisons plus fuel isotope evolution.

4 Detailed Code Walkthrough: reactorPhysics.py

This section provides a comprehensive line-by-line analysis of the reactor physics implementation.

4.1 Module Header and Physical Constants

```
#!/usr/bin/env python3

# Contains reactor kenetics equations
# and reactor parameters

import numpy as np

# 6-group delayed neutron precursor data for U-235 (more accurate)
beta_i = np.array([0.000215, 0.001424, 0.001274, 0.002568, 0.000748,
    0.000273])
# Delayed neutron fractions for each of the 6 groups
# These values are specific to U-235 thermal fission
# Units: dimensionless (fraction of total neutrons)

lambda_i = np.array([0.0124, 0.0305, 0.111, 0.301, 1.14, 3.01])
# Decay constants [1/s] for each precursor group
# These determine the time scales at which delayed neutrons are
    released
# Group 1: ~80 second half-life (long-lived precursors)
# Group 6: ~0.23 second half-life (short-lived precursors)

beta = np.sum(beta_i)
# Total delayed neutron fraction = 0.0065 (0.65%)
# This is the key parameter that makes reactor control possible
# Without delayed neutrons, reactors would be uncontrollable

Lamb = 10.e-5
# Average neutron lifetime (generation time) [s]
# For thermal reactors: ~10^-5 seconds
# This is the time from neutron birth to inducing next fission
```

Listing 1: Physical Constants and Delayed Neutron Data

Explanation: These constants define the fundamental nuclear data for U-235. The six-group delayed neutron model provides accurate representation of precursor decay dynamics across different time scales. The total delayed neutron fraction $\beta = 0.0065$ is critical—it determines the dollar-based reactivity scale and enables controllable reactor operation.

```
v = 2200.e3
# Neutron velocity cm/s
# This is the thermal neutron velocity at 20 C
# Corresponds to 0.025 eV neutron energy (thermal equilibrium)

Ef = 3.204e-11
# Energy per fission [J]
# For U-235: approximately 200 MeV per fission
# Converted to Joules: 200 MeV * 1.602e-13 J/MeV

Sigma_f = 0.0065
# Macrosopic fission cross section in reactor [1/cm]
# This depends on fuel enrichment and geometry
# Represents probability of fission per unit path length

Vr = 3.e6
# Reactor volumue [cc] = 3 cubic meters
```

```
# Typical small research reactor size
# Determines total power capacity and heat generation

Lc = Lamb * v
# Mean neutron travel length in core [cm]
# = (10^-5 s) * (2.2    10^6 cm/s) = 22 cm
# Characteristic diffusion length scale

VfFuel = 0.4
# Volume fraction occupied by fuel
# Remaining 60% is coolant, structure, etc.

VfH2O = 1. - VfFuel
# Volume fraction of water (coolant + moderator)
```

Listing 2: Reactor Geometry and Material Properties

Explanation: These parameters define the reactor geometry and material composition. The fission cross section, reactor volume, and volume fractions determine the neutron multiplication and power density. The 40% fuel volume fraction is typical of light water reactors.

```
hc = 1.
# W/cm^2 * K avg heat transfer coeff between fuel and water
# This is a simplified constant coefficient
# In reality, h depends on flow rate, temperature, geometry
# Typical values: 0.5-2 W/cm^2 K  for LWR conditions

Ac = 4.e5
# cm^2  fuel to coolant contact area
# Large surface area needed for efficient heat removal
# = 40 m^2 for 3 m^3 reactor (reasonable geometric estimate)

Tin = 450.
# K coolant inlet temperature
# = 177 C , typical for pressurized water reactors
# Below boiling point at atmospheric pressure
```

Listing 3: Heat Transfer Parameters

4.2 Reactivity Feedback Coefficient

```
alphaT = -0.007 * 1.e-5 / beta
# pcm / K / beta  reactivity per kelvin
# Negative temperature coefficient provides stability
# As fuel heats up, reactivity decreases (negative feedback)
# Magnitude: -0.007 pcm/K / 0.0065     -0.0011 $/K
# This self-regulating behavior is crucial for reactor safety

# Number of delayed neutron groups
NUM_GROUPS = 6
```

Listing 4: Temperature Coefficient of Reactivity

Physical Significance: The negative temperature coefficient is a cornerstone of reactor safety. As power increases, fuel temperature rises, which reduces reactivity, naturally limiting the power excursion. This provides inherent safety even without active control systems.

4.3 Neutron Population Dynamics

```python
def dndt(S, t, reactivity):
    """
    Time derivative of neutron population with 6 delayed neutron groups.

    Implements: dn/dt = [(rho - beta)/Lambda] * n + sum(lambda_i * C_i)

    Args:
        S: State vector [n, C1, C2, C3, C4, C5, C6, Tfuel, Tcoolant, h]
        t: Time (not explicitly used, but required by ODE solver)
        reactivity: Current reactivity in dollars

    Returns:
        dn/dt: Rate of change of neutron density [neutrons/cm^3/s]
    """
    # Sum contributions from all delayed neutron groups
    # Each group releases neutrons at rate lambda_i * C_i
    delayed_contribution = np.sum(lambda_i * S[1:7])

    # Main point kinetics equation
    # Prompt neutron contribution: (reactivity - beta) / Lambda * n
    # Delayed neutron contribution: sum of all precursor decay rates
    ndot = (reactivity - beta) / Lamb * S[0] + delayed_contribution

    # Prevent unphysical negative neutron population
    # If neutrons are already zero and decreasing, set derivative to zero
    if S[0] <= 0. and ndot < 0.:
        return 0.
    else:
        return ndot
```

Listing 5: Neutron Density Time Derivative

Mathematical Details: This function implements Equation (1) from the theory section. The prompt term $(\rho - \beta)/\Lambda$ has units of $[1/\text{s}]$, and when multiplied by neutron density gives $[\text{neutrons}/\text{cm}^3/\text{s}]$. The delayed contribution sums over all six precursor groups, each contributing at their characteristic decay rate λ_i.

4.4 Precursor Population Dynamics

```python
def dCdt(S, t, group_index):
    """
    Time derivative of delayed neutron precursor population for a specific group.

    Implements: dC_i/dt = (beta_i/Lambda) * n - lambda_i * C_i

    Args:
        S: State vector
        t: Time
        group_index: Which precursor group (0-5 for groups 1-6)

    Returns:
```

```
        dC_i/dt: Rate of change of precursor concentration [nuclei/cm^3/s]
    """
    C_index = group_index + 1
    # Precursor concentration stored at S[1] to S[6]
    # Index 0 is neutron density

    # Production of precursors from fissions
    # beta_i fraction of neutrons create this precursor type
    production = (beta_i[group_index] / Lamb) * S[0]

    # Loss of precursors due to radioactive decay
    # Releases delayed neutrons at rate lambda_i
    decay = lambda_i[group_index] * S[C_index]

    Cdot = production - decay

    # Prevent unphysical negative precursor concentrations
    if S[C_index] < 0. and Cdot < 0.:
        return 0.
    else:
        return Cdot
```

Listing 6: Delayed Neutron Precursor Evolution

Physics Note: The precursor production rate follows directly from the point kinetics equations. At steady state (criticality), the neutron balance requires $\rho = 0$, which is satisfied when precursor production equals decay. This ensures that a critical reactor displays zero reactivity in the simulation.

4.5 Thermal Power Calculation

```
def qFuel(n):
    """
    Given neutron population return thermal power

    Power [W] = Volume * fuel_fraction * (neutron_density * velocity)
                * fission_cross_section * energy_per_fission

    Args:
        n: Neutron density [neutrons/cm^3]

    Returns:
        Thermal power [W]
    """
    return Vr * VfFuel * (n * v) * Sigma_f * Ef
```

Listing 7: Fission Power Calculation

Physical Interpretation: This calculates power from neutron density using:

$$Q = V_r \cdot V_{f,fuel} \cdot (n \cdot v) \cdot \Sigma_f \cdot E_f \tag{28}$$

The product $(n \cdot v)$ gives neutron flux [neutrons/cm^2/s], multiplied by Σ_f gives fission rate density [fissions/cm^3/s], and multiplied by E_f gives power density [W/cm^3].

4.6 Fuel Temperature Dynamics

```python
def dTfdt(S, t, mdotC):
    """
    Time derivative of fuel temperature with improved heat transfer.
    Uses temperature-dependent properties and Dittus-Boelter correlation.

    Energy balance: Heat in (fission) = Heat out (to coolant) + Heat stored

    Args:
        S: State vector
        t: Time
        mdotC: Coolant mass flow rate [g/s]

    Returns:
        dT_fuel/dt [K/s]
    """
    # Temperature-dependent UO2 heat capacity (J/g*K)
    T_fuel = S[7]  # fuel temperature at index 7
    CpUO2 = 0.2455 + 5.86e-5 * (T_fuel - 273.15)
    # Linear temperature dependence for UO2
    # Base value ~245 J/kg K at room temp
    # Increases slightly with temperature

    densityUO2 = 12.5  # g/cc (theoretical density ~10.97, using higher for sintered fuel)

    # Improved heat transfer coefficient using flow-dependent correlation
    # Dittus-Boelter-like correlation: h = h0 * (mdot/mdot0)^0.8
    h0 = 1.5  # W/cm^2*K baseline heat transfer coefficient
    mdot0 = 1000.e3  # reference flow rate [g/s] = 1000 kg/s
    h = h0 * (mdotC / mdot0) ** 0.8
    # Heat transfer improves with flow rate (turbulent convection)
    # Power 0.8 from Dittus-Boelter correlation for turbulent flow

    # Heat generation from fission
    heat_in = qFuel(S[0])

    # Heat removal to coolant
    heat_out = Ac * h * (S[7] - S[8])  # Newton's law of cooling

    # Fuel thermal mass
    fuel_mass = densityUO2 * VfFuel * Vr

    # Energy balance: dE/dt = Q_in - Q_out
    # dE/dt = m * Cp * dT/dt
    return (heat_in - heat_out) / (fuel_mass * CpUO2)
```

Listing 8: Fuel Temperature Evolution with Improved Heat Transfer

Engineering Details: The Dittus-Boelter correlation $h \propto \dot{m}^{0.8}$ captures the turbulent flow heat transfer enhancement. Higher coolant flow rates improve heat removal, reducing fuel temperatures and preventing overheating. The temperature-dependent heat capacity makes the model more accurate at high temperatures.

4.7 Coolant Temperature Dynamics

```python
def dTcdt(S, t, mdotC):
    """
    Time derivative of water coolant with improved heat transfer.
    Uses temperature-dependent properties.

    Energy balance for coolant:
    Heat in (from fuel) + Heat in (from inlet flow) = Heat out (exit flow) + Heat stored

    Args:
        S: State vector
        t: Time
        mdotC: Coolant mass flow rate [g/s]

    Returns:
        dT_coolant/dt [K/s]
    """
    # Temperature-dependent water properties
    T_coolant = S[8]  # coolant temperature at index 8
    CpH2O = 4.2 - 0.0005 * (T_coolant - 273.15)
    # Water heat capacity in J/g*K
    # Decreases slightly with temperature (accurate for liquid water)
    # At 25 C : ~4.18 J/g K

    densityH2O = 1.0  # g/cc (simplified, actual density varies with T)

    # Use same improved heat transfer coefficient
    h0 = 1.5
    mdot0 = 1000.e3
    h = h0 * (mdotC / mdot0) ** 0.8

    # Heat transferred from fuel to coolant
    heat_from_fuel = Ac * h * (S[7] - S[8])

    # Energy brought in by inlet coolant flow
    # mdotC * Cp * (Tin - Tcoolant)
    # If coolant is hotter than inlet, this term is negative (cooling effect)
    heat_from_inlet = CpH2O * (Tin - S[8]) * mdotC

    # Coolant thermal mass
    coolant_mass = densityH2O * Vr * CpH2O

    return (heat_from_fuel + heat_from_inlet) / coolant_mass
```

Listing 9: Coolant Temperature Evolution

Physical Insight: The coolant acts as a heat transport medium. Heat is transferred from the hot fuel (first term) and cold inlet coolant continuously refreshes the core (second term). Higher flow rates increase the cooling effect, as seen in the `heat_from_inlet` term being proportional to $\dot{m}_c$.

4.8 Control Rod Worth Functions

```python
def diffRodWorth(h):
```

```
    """
    Improved differential control rod worth curve using cosine shape.
    Tuned to achieve total worth of 0.1 $ from fully inserted to fully withdrawn.
    h is fractional height: h=0 is fully inserted, h=100 is fully withdrawn
    delta_h * R(h) = reactivity change

    Physical basis: Rod worth follows importance-weighted distribution
    Maximum near core midplane (h=50), zero at top/bottom

    Args:
        h: Rod height position [%] (0 = inserted, 100 = withdrawn)

    Returns:
        Differential rod worth [$/% height]
    """
    scalingFac = 0.01021 * 1.e-5 / beta
    # Tuning factor to achieve desired total rod worth
    # Result: total integral worth = 0.1 dollars

    # Sinusoidal shape: peaks at h=50 (core midplane)
    return scalingFac * np.sin(np.pi * h / 100.0) * 100.0

def intRodWorth(h1, h2):
    """
    Integral control rod worth curve.
    Returns reactivity in dollars ($/Beta)

    Integrates differential worth from position h1 to h2
    rho = integral from h1 to h2 of R(h) dh

    Args:
        h1: Initial rod position [%]
        h2: Final rod position [%]

    Returns:
        Reactivity change [dollars]
    """
    scalingFac = 0.01021 * 1.e-5 / beta

    # Analytical integral of sin(pi*h/100):
    # integral = -100/pi * cos(pi*h/100)
    integral = lambda h: -100.0 * scalingFac * (100.0 / np.pi) * np.cos(np.pi * h / 100.0)

    return (integral(h2) - integral(h1))
```

Listing 10: Control Rod Reactivity Worth Curves

Mathematical Justification: The sinusoidal rod worth shape reflects neutron importance weighting in the reactor core. Neutrons near the core center (where flux is maximum) have the highest importance for criticality. Therefore, control rod position changes near the midplane have maximum reactivity effect.

4.9 Total Reactivity Calculation

```python
def rho(S, t, hrate, deltaT):
    """
    Temperature and control rod reactivity.
    Reactivity in units of Dollars  (deltaK / Beta)
    Takes control rod movement rate in (%/s)

    Total reactivity = control rod component + temperature feedback

    Args:
        S: State vector
        t: Time
        hrate: Rod movement rate [%/s] (not used in calculation, but kept for interface)
        deltaT: Time step (not used, but kept for interface)

    Returns:
        Total reactivity [dollars]
    """
    # Temperature feedback: negative for stability
    # Alpha_T * (T_fuel - T_inlet)
    temp_reactivity = alphaT * (S[7] - Tin)

    # Control rod reactivity: integral worth from 0 to current position
    # More withdrawn (higher h) = more positive reactivity
    rod_reactivity = intRodWorth(0., S[9])

    return temp_reactivity + rod_reactivity
```

Listing 11: Combined Reactivity from All Feedback Mechanisms

Design Philosophy: Separating reactivity into components (temperature and control) allows clear understanding of competing effects. During a power increase, positive rod reactivity may be countered by negative temperature feedback, demonstrating the self-regulating nature of the design.

4.10 Complete Reactor System ODEs

```python
def reactorSystem(S, t, hrate, deltaT, mdotC=1000.e3):
    """
    Complete reactor system with 6 delayed neutron groups.
    State vector S = [n, C1, C2, C3, C4, C5, C6, Tfuel, Tcoolant, rodPosition]

    This function returns dS/dt for the entire 10-dimensional state vector.
    Called by scipy's ODE integrator (odeint).

    Args:
        S: Current state vector (length 10)
        t: Current time [s]
        hrate: Control rod movement rate [%/s]
        deltaT: Time step (for reference, not used in continuous ODEs)
        mdotC: Coolant mass flow rate [g/s]

    Returns:
        dS/dt: Time derivatives of all state variables
    """
```

```
    # Calculate current reactivity based on state
    reactivity = rho(S, t, hrate, deltaT)

    # Build derivative vector
    # Start with neutron population derivative
    dSdt = [dndt(S, t, reactivity)]

    # Add all 6 precursor group derivatives
    for i in range(NUM_GROUPS):
        dSdt.append(dCdt(S, t, i))

    # Add temperature derivatives (fuel and coolant)
    dSdt.append(dTfdt(S, t, mdotC))
    dSdt.append(dTcdt(S, t, mdotC))

    # Add control rod position derivative
    # Rod position changes at specified rate
    dSdt.append(hrate)

    return dSdt
```

Listing 12: Full Reactor System Differential Equations

Numerical Integration: This function returns $\frac{d\mathbf{S}}{dt}$ for the entire state vector. The scipy `odeint` function uses this to advance the system forward in time using adaptive step size Runge-Kutta methods, ensuring numerical stability and accuracy.

5 Reactor Control System: reactor.py

The `reactor.py` module provides the high-level interface for reactor operation, implementing control logic, safety systems, and state management.

5.1 DUNEReactor Class Structure

```
class DUNEReactor(object):
    """
    Provides methods to interact with the point kenetics model.
    The reactor system state vector (with 6 delayed neutron groups):
    S = [neutrons/cc, C1, C2, C3, C4, C5, C6, fuelT, coolantT,
   rodPosition]
    """
    def __init__(self, initialSystemState=None, tstep=0.01):
        """ Initialize reactor system state """
        if initialSystemState is None:
            # Default initial conditions
            n0 = 5.e7  # Initial neutron density [neutrons/cm^3]
            # Corresponds to ~1 MW thermal power at steady state

            # Initialize precursor concentrations
            # At equilibrium: C_i = (beta_i / lambda_i / Lambda) * n
            # Simplified here: distribute total beta equally among
   groups
            C_init = [n0 * 0.0065 / 6.0] * 6

            # Complete initial state:
```

```
            # [n, C1-C6, Tfuel, Tcoolant, rodPosition]
            initialSystemState = [n0] + C_init + [450., 450., 0.]
            # Tfuel = Tcoolant = 450 K (isothermal start)
            # Rod position = 0% (fully inserted, subcritical)

        self.S = np.array(initialSystemState)
        self.reactivity = rho(self.S, 0, 0, 0)
        self.tstep = tstep  # Time step for integration [s]
        self.t = np.array([0, self.tstep])

        # Control variables
        self.hrate = 0.0  # Rod movement rate [% / s]
        self.rodSetPoint = 0.0  # Desired rod position [%]
        self.mdotC = 1000.e3  # Coolant flow rate [g / s] = 1000 kg/s
        self.coolantSetPoint = 1000.e3
        self.pwrCtrl = False  # Power control mode off by default
        self.scramToggle = False  # SCRAM status

        # Storage for plotting
        self.maxTime = 100.  # Store last 100 seconds of data
        dataStorLength = int(self.maxTime / self.tstep)
        self.time = np.zeros(dataStorLength)
        # Store: [n, sum(C1-C6), Tfuel, Tcoolant, rodPosition]
        self.storVals = np.zeros((5, dataStorLength))
```

Listing 13: DUNEReactor Class Initialization

Design Rationale: The class encapsulates all reactor state and control parameters. The rolling data storage (last 100 seconds) provides efficient memory usage while maintaining sufficient history for visualization and trend analysis.

5.2 Time Stepping and Integration

```
def timeStep(self):
    """ Step reactor system forward in time """
    # Pre-step checks: control logic, safety systems
    self.__preStep()

    # Integrate ODEs from t to t+tstep
    # odeint returns array of solutions; take the last (final) value
    self.S = integrate.odeint(reactorSystem, self.S, self.t,
                              args=(self.hrate, self.tstep, self.mdotC)
    )[-1]

    # Update reactivity based on new state
    self.reactivity = rho(self.S, 0, 0, 0)

    # Advance time
    self.t += self.tstep

    # Update rolling data storage
    # Shift arrays left (discard oldest), append new values
    self.storVals = np.roll(self.storVals, -1, axis=1)
    self.time = np.roll(self.time, -1)
    self.time[-1] = self.t[-1]

    # Store key values for plotting
```

```
    self.storVals[:, -1] = np.array([
        self.S[0],            # Neutron density
        np.sum(self.S[1:7]), # Total precursor concentration
        self.S[7],            # Fuel temperature
        self.S[8],            # Coolant temperature
        self.S[9]             # Rod position
    ])
```

Listing 14: Time Integration Method

Numerical Method: The `scipy.integrate.odeint` function uses LSODA (Livermore Solver for Ordinary Differential equations with Automatic method switching), which automatically switches between stiff and non-stiff integration methods. This is essential for reactor kinetics, which can exhibit stiff behavior during rapid transients.

5.3 Pre-Step Control Logic

```
def __preStep(self):
    """
    Check for valid rod movements or SCRAM condition
    Applied before each time integration step
    """
    # Determine rod movement rate based on control mode
    if self.pwrCtrl:
        # Automatic power control using PID
        self.__controlPID()
    else:
        # Manual rod positioning control
        self.__rodCtrl()

    # Enforce physical rod position limits
    if self.hrate < 0 and self.S[9] <= 0.:
        # Cannot insert beyond 0% (fully in)
        self.hrate = 0.
    elif self.hrate > 0 and self.S[9] >= 100.:
        # Cannot withdraw beyond 100% (fully out)
        self.hrate = 0.

    # Update coolant flow rate (with ramping)
    self.__controlCoolantRate()

    # Check for SCRAM conditions
    self.__scramCheck()

    # If SCRAM active, immediately insert rods
    if self.scramToggle:
        self.S[9] = 0.  # Rods fully inserted
        self.hrate = 0.  # No further movement
```

Listing 15: Control Logic and Safety Checks Before Each Time Step

5.4 SCRAM Protection System

```
def __scramCheck(self):
    """
    Check for conditions which require us to SCRAM
```

```
    Implements reactor protection system (RPS)
    """
    if self.S[7] > 1700:
        # Fuel temperature SCRAM setpoint exceeded
        # 1700 K = 1427 C  (below UO2 melting point ~2865 C )
        # Conservative limit to prevent fuel damage
        print("Fuel Temperature SCRAM setpoint Exceeded")
        self.SCRAM()

    elif self.S[8] > 700:
        # Coolant temperature SCRAM setpoint exceeded
        # 700 K = 427 C  (well above water boiling point)
        # Prevents loss of coolant and core damage
        print("Coolant Temperature SCRAM setpoint Exceeded")
        self.SCRAM()
    else:
        # All parameters within normal operating range
        pass

def SCRAM(self, scramToggle=True):
    """
    You crashed the reactor.
    Initiate emergency shutdown by rapid rod insertion.
    """
    self.scramToggle = scramToggle
```

Listing 16: Automatic Reactor SCRAM Logic

Safety Philosophy: The SCRAM system provides defense-in-depth protection. Multiple parameters are monitored, and any limit violation triggers immediate shutdown. The fuel temperature limit (1700 K) is set conservatively below the UO_2 melting point to ensure core integrity even under transient conditions.

5.5 Manual Rod Control

```
def __rodCtrl(self):
    """
    Manual control rod positioning mode
    Smoothly moves rods toward setpoint using tanh function
    """
    # Calculate position error
    diff = self.S[9] - self.rodSetPoint

    # Apply tanh function for smooth approach
    # tanh provides fast initial movement that slows near target
    fnDiff = np.tanh(1.0 * abs(diff))

    if diff < 0.:
        # Current position below setpoint: withdraw rods (increase h)
        self.hrate = 0.5 * fnDiff  # Maximum 0.5%/s withdrawal
    elif diff > 0.:
        # Current position above setpoint: insert rods (decrease h)
        self.hrate = -0.5 * fnDiff  # Maximum 0.5%/s insertion
    else:
        # At setpoint: no movement
        self.hrate = 0.

```

```
def setRodPosition(self, rodPos):
    """Set desired rod position [%]"""
    self.rodSetPoint = rodPos

def setRodRate(self, rodRate):
    """Directly set rod movement rate [%/s] (manual override)"""
    if not self.pwrCtrl:
        self.hrate = rodRate
```

Listing 17: Smooth Rod Position Control with Tanh Relaxation

Control Strategy: The hyperbolic tangent function provides smooth, stable rod movement. As the rod approaches the setpoint, the movement rate automatically decreases, preventing overshoot and oscillation. This mimics realistic mechanical control systems with position servos.

5.6 PID Power Control

```
def togglePwrCtrl(self, pwrSet, pwrCtrlToggle=True):
    """
    Enable/disable automatic power control
    Set desired power in MW
    """
    self.pwrSet = pwrSet  # Target power [MW]
    self.pwrCtrl = pwrCtrlToggle
    self.pidBias = 0.0  # PID integrator bias
    self.hrate = 0.0  # Reset rod movement

def __controlPID(self):
    """
    PID controller for automatic power regulation
    Adjusts control rod position to maintain target power
    """
    maxRate = 0.60  # Maximum rod movement rate [%/s]

    # PID tuning parameters (manually tuned for stability)
    Kp = 0.0100000   # Proportional gain
    Ki = 0.0001000   # Integral gain
    Kd = 0.0001000   # Derivative gain

    # Current power level
    currentpwr = qFuel(self.S[0]) / 1.e6  # [MW]

    # Error signal: desired - actual
    errorFn = self.pwrSet - qFuel(self.storVals[0, :]) / 1.e6

    # Integral term: sum of recent errors
    # Use last 100 stored values for integral accumulation
    errorIntegral = np.sum(errorFn[-100:])

    # Derivative term: rate of error change
    errorDerivative = (errorFn[-1] - errorFn[-2]) / self.tstep

    if hasattr(self, 'pwrSet'):
        # PID control law
        pidOut = (self.pidBias +
                  Kp * (self.pwrSet - currentpwr) +  # Proportional
```

```
                    Ki * errorIntegral +                # Integral
                    Kd * errorDerivative)               # Derivative

        self.hrate = pidOut

        # Limit rod movement rate to physical constraints
        if abs(self.hrate) > maxRate:
            # Preserve sign, limit magnitude
            self.hrate = maxRate * (self.hrate / abs(self.hrate))
    else:
        # Initialize with current power as setpoint
        self.togglePwrCtrl(qFuel(self.S[0]) / 1.e6)
```

Listing 18: Proportional-Integral-Derivative Power Controller

Control Theory: The PID controller implements three actions:

- **Proportional (P)**: Responds to current error magnitude. Larger errors produce faster rod movement.
- **Integral (I)**: Eliminates steady-state error by accumulating past errors. Ensures the reactor reaches exactly the target power.
- **Derivative (D)**: Anticipates future error by responding to error rate. Provides damping to prevent overshoot.

The relatively small gains ($K_p = 0.01$, $K_i = 0.0001$, $K_d = 0.0001$) are necessary because reactor power is extremely sensitive to rod position. Aggressive tuning would cause instability and power oscillations.

5.7 Coolant Flow Control

5.7.1 Dynamic Flow Rate Adjustment

The reactor automatically adjusts coolant flow rate based on power output when manual control is disabled:

```
def __updateCoolantForPower(self):
    """
    Automatically adjust coolant setpoint based on reactor power
    Maps power to coolant flow rate: low power ~ 200 kg/s, high power ~
    1200 kg/s
    """
    currentPower = qFuel(self.S[0]) / 1.e6  # Power in MW
    maxPwr = 600.  # Maximum power for scaling

    # Normalize power (0 to 1)
    normPwr = abs(currentPower / maxPwr)
    if normPwr > 1.0:
        normPwr = 1.0

    # Map to coolant flow rate range
    minFlowRate = 200.e3  # 200 kg/s = 200000 g/s at minimum power
    maxFlowRate = 1200.e3  # 1200 kg/s = 1200000 g/s at maximum power
    self.coolantSetPoint = minFlowRate + (maxFlowRate - minFlowRate) *
   normPwr
```

Listing 19: Automatic Power-Based Flow Rate Adjustment

Engineering Rationale: In real PWR reactors, coolant flow rate must match thermal power output to maintain proper heat removal and prevent departure from nucleate boiling (DNB). The optimized linear mapping from 200 kg/s to 1200 kg/s provides adequate cooling margin across the entire power range while reducing pumping power requirements at maximum capacity.

5.7.2 Manual Coolant Control Mode

```
def toggleCoolantCtrl(self, coolantSet, coolantCtrlToggle=True):
    """
    Set coolant flow rate in kg/s (converts to g/s internally)
    When enabled, user has direct control over flow rate
    """
    self.coolantSetPoint = coolantSet * 1.e3  # convert kg/s to g/s
    self.coolantCtrl = coolantCtrlToggle
    if self.coolantCtrl:
        self.mdotC = self.coolantSetPoint
```

Listing 20: User-Controlled Flow Rate Mode

5.7.3 Gradual Flow Rate Ramping

```
def setCoolantRate(self, mdotCin):
    """Set desired coolant flow rate [g/s]"""
    self.coolantSetPoint = mdotCin

def __controlCoolantRate(self):
    """
    Gradually adjust coolant flow rate toward setpoint
    Prevents sudden flow changes that could cause thermal shock
    """
    # Calculate flow rate error
    diff = (self.coolantSetPoint - self.mdotC) / 10.

    # Apply tanh for smooth ramping
    fnDiff = np.tanh(1.0 * abs(diff))

    if self.coolantSetPoint > self.mdotC:
        # Increase flow rate
        self.mdotC += 1. / self.tstep * fnDiff
    elif self.coolantSetPoint < self.mdotC:
        # Decrease flow rate
        self.mdotC -= 1. / self.tstep * fnDiff
    else:
        pass
```

Listing 21: Smooth Coolant Flow Transitions

Engineering Consideration: Gradual flow rate changes prevent thermal shock to reactor components. Sudden coolant flow changes could cause rapid temperature gradients, potentially damaging fuel cladding or pressure boundaries. The tanh function provides smooth, asymptotic approach to the setpoint.

5.7.4 Prompt Jump Mode

```
def togglePromptJumpMode(self, promptCriticalToggle=True):
    """
    Toggle Prompt Jump Mode.
    When enabled, instantly inserts ~$0.003 reactivity by quickly
    withdrawing the control rod (similar to reverse SCRAM).
    Automatic SCRAM remains enabled for safety.
    WARNING: This is for educational demonstration only!
    """
    self.promptCriticalMode = promptCriticalToggle
    if promptCriticalToggle:
        # Instantly withdraw rod by ~3% to insert ~$0.003 reactivity
        # (with $0.2 total rod worth, 3% withdrawal gives ~$0.003)
        newPos = min(self.S[9] + 3.0, 100.0)
        self.S[9] = newPos
        self.hrate = 0.0  # Stop any ongoing rod movement
```

Listing 22: Prompt Jump Mode for Reactivity Insertion Demonstration

Educational Purpose: Prompt Jump Mode allows students to observe the characteristic rapid power increase that occurs when positive reactivity is suddenly inserted. Unlike real prompt criticality accidents ($\rho > \$1.00$), this mode inserts a small, safe amount of reactivity ($\sim \$0.003$) to demonstrate the phenomenon without risking simulated core damage. Automatic SCRAM protection remains active, ensuring the reactor can respond to any temperature excursions.

5.8 Testing and Validation

```
def test():
    """
    Test reactor in rod control and power control modes.
    Validates basic functionality before GUI integration.
    """
    i = 0
    t0 = time.time()  # Wall clock timing
    duneReactor = DUNEReactor()

    # ===== TEST 1: Manual Rod Control =====
    duneReactor.setRodPosition(50.)  # Withdraw rods to 50%
    while i < 10000:
        duneReactor.timeStep()
        print("====================================")
        print("Time [s] = %f" % duneReactor.t[-1])
        print("Rod percent Withdrawn = %f" % duneReactor.S[9])
        print("Reactor Power [MW] = %f " % float(qFuel(duneReactor.S
    [0]) / 1.e6))
        print("Tfuel [K] = %f ,  Tcoolant [K] = %f" % (duneReactor.S
    [7], duneReactor.S[8]))
        i += 1

    # ===== TEST 2: Automatic Power Control =====
    i = 0
    duneReactor.togglePwrCtrl(200.)  # Set target power to 200 MW
    while i < 10000:
        duneReactor.timeStep()
        print("====================================")
        print("Time [s] = %f" % duneReactor.t[-1])
        print("Rod percent Withdrawn = %f" % duneReactor.S[9])
```

```
29          print("Reactor Power [MW] = %f " % float(qFuel(duneReactor.S
    [0]) / 1.e6))
30          print("Tfuel [K] = %f ,  Tcoolant [K] = %f" % (duneReactor.S
    [7], duneReactor.S[8]))
31          i += 1
32
33      t1 = time.time()
34      print("Execution time: %f seconds" % (t1 - t0))
35
36  if __name__ == "__main__":
37      test()
```

Listing 23: Reactor Test Function

6 GUI Frontend: duneReactor.py

The graphical user interface provides interactive control and real-time visualization of reactor behavior.

6.1 GUI Architecture Overview

The GUI is built using:

- **wxPython**: Cross-platform GUI framework for controls and layout
- **matplotlib**: Embedded plotting for real-time data visualization
- **Timer events**: Asynchronous reactor updates and plot refreshes
- **Serial communication**: Optional Arduino integration for hardware feedback

6.2 Main Application Class

```
1  class CalcFrame(gui.MyFrame1):
2      """
3      Main application frame inheriting from auto-generated GUI template
4      Implements all event handlers and control logic
5      """
6      def __init__(self, parent):
7          # Initialize parent class (GUI layout)
8          gui.MyFrame1.__init__(self, parent)
9
10         # Set initial conditions
11         self.setInitConds()
12
13         # Initialize Arduino serial connection (if available)
14         self.ser = initSerial()
15
16         # Create reactor simulation instance
17         self.legoReactor = rct.DUNEReactor(tstep=0.005)
18         # Small time step (5 ms) for smooth real-time operation
19
20         # Generate initial data
21         self.duneReactor.timeStep()
```

```
        self.data = [self.duneReactor.time, self.duneReactor.storVals]

        # Initialize coolant flow display
        self.coolantBox.SetValue(str(round(self.duneReactor.mdotC / 1.
    e3, 2)))

        # Setup matplotlib plotting panel
        self.create_plot_panel()

        # Setup timers for simulation and visualization
        self.recalc_timer = wx.Timer(self)
        self.redraw_timer = wx.Timer(self)
        self.Bind(wx.EVT_TIMER, self.on_recalc_timer, self.recalc_timer
    )
        self.Bind(wx.EVT_TIMER, self.on_redraw_timer, self.redraw_timer
    )

        # Start timers
        # Recalculate every 2 ms (500 Hz update rate)
        self.recalc_timer.Start(2)
        # Redraw plot every 1000 ms (1 Hz refresh rate)
        self.redraw_timer.Start(1000)
        # High simulation rate ensures smooth control response
        # Lower plot refresh reduces GUI overhead
```

Listing 24: CalcFrame Class: Main Application Window

Design Pattern: The separation between calculation rate (500 Hz) and redraw rate (1 Hz) optimizes performance. The reactor physics updates rapidly for accurate control, while expensive plot rendering occurs less frequently to maintain GUI responsiveness.

6.3 Plot Initialization and Configuration

```
def init_plot(self):
    """
    Initialize matplotlib figure and axes
    Creates four subplots: power, reactivity, temperature, and Xenon
    -135
    """
    self.dpi = 100
    self.fig = Figure((16.0, 10.0), dpi=self.dpi)  # Large full-screen
    figure

    # 2x2 subplot grid
    self.axes1 = self.fig.add_subplot(221)  # Top left: Power [MW]
    self.axes4 = self.fig.add_subplot(222)  # Top right: Reactivity ($)
    self.axes2 = self.fig.add_subplot(223)  # Bottom left: Temperatures
    self.axes3 = self.axes2.twinx()  # Dual y-axis for fuel/coolant
    temps
    self.axes5 = self.fig.add_subplot(224)  # Bottom right: Xenon-135

    # Set background colors
    if LooseVersion(matplotlib.__version__) >= LooseVersion('2.0.0'):
        self.axes1.set_facecolor('white')
        self.axes4.set_facecolor('white')
        self.axes5.set_facecolor('white')
    else:
```

```
        self.axes1.set_axis_bgcolor('white')
        self.axes4.set_axis_bgcolor('white')
        self.axes5.set_axis_bgcolor('white')

    self.axes1.set_title('Reactor Power [MW] Trace', size=12)
    pylab.setp(self.axes1.get_xticklabels(), fontsize=8)
    pylab.setp(self.axes1.get_yticklabels(), fontsize=8)
    pylab.setp(self.axes4.get_xticklabels(), fontsize=8)
    pylab.setp(self.axes4.get_yticklabels(), fontsize=8)
    pylab.setp(self.axes5.get_xticklabels(), fontsize=8)
    pylab.setp(self.axes5.get_yticklabels(), fontsize=8)

def create_plot_panel(self):
    """Embed matplotlib canvas in wxPython panel"""
    self.init_plot()
    self.canvas = FigCanvas(self.m_panel2, -1, self.fig)
    # Maximize window to full screen on startup
    self.Maximize(True)
```

Listing 25: Enhanced Matplotlib Plot Setup with Four Panels

GUI Enhancement: The updated layout provides comprehensive real-time visualization with four distinct monitoring panels, enabling operators to simultaneously track power output, reactivity balance (including Xenon poisoning contribution), thermal conditions, and fission product concentrations. The full-screen mode maximizes data visibility for educational demonstrations.

6.4 Real-Time Plotting

```
def draw_plot(self):
    """
    Update plots with latest reactor data
    Implements zoom functionality and dual-axis temperature plotting
    """
    # Determine plot window size based on zoom level
    zoomPercentage = self.zoom / 100.
    if zoomPercentage < 0.02:
        zoomPercentage = 0.02  # Minimum 2% zoom (2 seconds)

    # Calculate number of data points to display
    plotMask = int(zoomPercentage * len(self.data[0]))

    # Create time axis
    xdata = np.array(np.array(range(plotMask)) / float(plotMask)) * \
            self.duneReactor.maxTime * zoomPercentage

    # Extract data traces
    pwrdata = qFuel(self.data[1][0, :][-plotMask:]) / 1.e6  # Power [MW
   ]
    fuelTdata = self.data[1][2, :][-plotMask:]  # Fuel temp [K]
    coolTdata = self.data[1][3, :][-plotMask:]  # Coolant temp [K]

    # Clear previous plots
    self.axes1.clear()
    self.axes2.clear()
    self.axes3.clear()

```

```
    # Set axis limits
    self.axes1.set_ylim(0, 650.)  # Power: 0-650 MW
    self.axes2.set_ylim(400, 1700.)  # Fuel temp: 400-1700 K
    self.axes3.set_ylim(400, 700.)  # Coolant temp: 400-700 K

    # Plot power
    self.axes1.set_title('Reactor Power [MW] Trace', size=12)
    self.axes1.set_ylabel('Power [MW]')
    self.axes1.set_xlabel(str(round(max(xdata), 0)) + ' time [s]')
    self.axes1.plot(xdata, pwrdata, linewidth=2, color='blue')

    # Plot temperatures on dual y-axes
    self.axes2.set_ylabel('Fuel Temperature [K]')
    self.axes3.set_ylabel('Coolant Temperature [K]')
    self.axes3.yaxis.set_label_position('right')
    self.axes3.yaxis.tick_right()

    # Red line for fuel temperature (left axis)
    fuelPlot, = self.axes2.plot(xdata, fuelTdata, color='r',
                                linewidth=2, label='Fuel T')
    # Blue line for coolant temperature (right axis)
    coolPlot, = self.axes3.plot(xdata, coolTdata, color='b',
                                linewidth=2, label='Coolant T')

    # Add legends
    handles, labels = self.axes2.get_legend_handles_labels()
    self.axes2.legend(handles, labels, loc=2)  # Upper left
    handles, labels = self.axes3.get_legend_handles_labels()
    self.axes3.legend(handles, labels, bbox_to_anchor=(0.402, 0.85))

    # Render updated plot
    self.canvas.draw()
```

Listing 26: Dynamic Plot Updating with Zoom Control

Visualization Strategy: The dual y-axis temperature plot allows simultaneous viewing of fuel and coolant temperatures despite their different ranges (fuel: 400-1700 K, coolant: 400-700 K). This is critical for understanding thermal coupling and heat transfer dynamics.

6.5 Event Handlers: User Interaction

```
def pauseSim(self, event):
    """Pause/unpause simulation"""
    self.paused = not self.paused

def SCRAM(self, event):
    """SCRAM button: toggle reactor shutdown"""
    self.scramToggle = not self.scramToggle
    self.duneReactor.SCRAM(bool(self.scramToggle))

def pwrCtrlON(self, event):
    """Toggle automatic power control mode"""
    pwrSet = self.pwrSetPt.GetValue()
    self.pwrCtrlToggle = not self.pwrCtrlToggle
    self.duneReactor.togglePwrCtrl(float(pwrSet), bool(self.
   pwrCtrlToggle))
```

```
15
16 def setReactorPwr(self, event):
17     """Update power setpoint (when in power control mode)"""
18     pwrSet = self.pwrSetPt.GetValue()
19     if self.pwrCtrlToggle:
20         self.duneReactor.togglePwrCtrl(float(pwrSet))
21
22 def setRodPos(self, event):
23     """Set rod position from text entry"""
24     enteredVal = self.rodSetPt.GetValue()
25     self.duneReactor.setRodPosition(float(enteredVal))
26     # Update slider to match
27     self.rodSlide.SetValue(100 - int(enteredVal))
28
29 def rodSlideSet(self, event):
30     """Set rod position from slider"""
31     # Slider is inverted: top=0% (inserted), bottom=100% (withdrawn)
32     self.rodSetPt.SetValue(str(100 - self.rodSlide.GetValue()))
33     self.duneReactor.setRodPosition(float(self.rodSetPt.GetValue()))
34
35 def setPlotZoom(self, event):
36     """Adjust plot time window zoom level"""
37     self.zoom = int(self.plotZoom.GetValue())
38
39 def coolantSet(self, event):
40     """Set coolant flow rate from text entry"""
41     coolantSet = self.coolantBox.GetValue()
42     if self.coolantCtrlToggle:
43         self.duneReactor.toggleCoolantCtrl(float(coolantSet))
44     else:
45         self.duneReactor.setCoolantRate(float(coolantSet) * 1.e3)
46
47 def coolantCtrlON(self, event):
48     """Toggle coolant flow control mode"""
49     coolantSet = self.coolantBox.GetValue()
50     self.coolantCtrlToggle = not self.coolantCtrlToggle
51     self.duneReactor.toggleCoolantCtrl(float(coolantSet),
52                                        bool(self.coolantCtrlToggle))
53
54 def PromptJumpON(self, event):
55     """Toggle Prompt Jump Mode for reactivity insertion demonstration""
   "
56     self.promptCriticalToggle = not self.promptCriticalToggle
57     self.duneReactor.togglePromptJumpMode(bool(self.
   promptCriticalToggle))
58     if self.promptCriticalToggle:
59         print("WARNING: Prompt Jump Mode ACTIVATED - Inserting ~$0.003
   reactivity")
60     else:
61         print("Prompt Jump Mode DEACTIVATED")
```

Listing 27: GUI Event Handler Implementations

6.6 Monitor Updates

```
1 def updateMonitors(self):
2     """
```

```
    Update all text displays and gauges with current reactor parameters
    Called every redraw cycle (1 Hz)
    """
    # Rod position display
    self.rodPosOut.SetValue(str(round(self.duneReactor.S[9], 1)))

    # Temperature displays
    self.cooltOut.SetValue(str(round(self.duneReactor.S[8], 2)))
    self.fueltOut.SetValue(str(round(self.duneReactor.S[7], 2)))

    # Power display
    self.powOut.SetValue(str(round(float(qFuel(self.duneReactor.S[0]) /
    1.e6), 6)))

    # Visual rod position gauge
    self.rodGauge.SetValue(int(self.duneReactor.S[9]))

    # Reactivity display (in dollars)
    self.reactivityOut.SetValue('{:.6f}'.format(self.duneReactor.
   reactivity))

    # Xenon-135 and Samarium-149 displays (scientific notation)
    self.xenonOut.SetValue('{:.3e}'.format(self.duneReactor.S[11]))
    self.samariumOut.SetValue('{:.3e}'.format(self.duneReactor.S[14]))

    # Coolant flow rate - only update if not in manual control mode
    if not self.coolantCtrlToggle:
        # Display current reactor coolant flow rate in kg/s
        self.coolantBox.SetValue(str(round(self.duneReactor.mdotC / 1.
   e3, 2)))
```

Listing 28: Update Display Monitors with Current State

Note: The coolant flow rate display is only updated automatically when manual control is disabled. This prevents the display from overwriting user input when the operator is manually adjusting flow rate. The reactivity display shows the current net reactivity in dollars ($), which should be near zero for a critical reactor at steady-state power.

6.7 CSV Data Logging System

The system automatically logs all critical reactor parameters to timestamped CSV files for post-simulation analysis.

6.7.1 CSV Initialization

```
def initCSVLogging(self):
    """Initialize CSV file for logging simulation data"""
    #Create SimulationData folder if it doesn't exist
    self.csv_folder = "SimulationData"
    if not os.path.exists(self.csv_folder):
        os.makedirs(self.csv_folder)

    # Create CSV filename with current date and time
    timestamp = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    self.csv_filename = os.path.join(self.csv_folder,
```

```
                                        f"reactor_sim_{timestamp}.csv")

    # Open CSV file and write header
    self.csv_file = open(self.csv_filename, 'w', newline='')
    self.csv_writer = csv.writer(self.csv_file)
    self.csv_writer.writerow(['Time(s)', 'Neutron_Density(#/cc)', '
Power(MW)',
                              'Reactivity($)', 'Fuel_Temp(K)', '
Coolant_Temp(K)',
                              'Flow_Rate(kg/s)', 'Rod_Position(%)'])
    self.last_log_time = 0.0
    self.log_interval = 0.5  # Log every 0.5 seconds
    print(f"CSV logging initialized: {self.csv_filename}")
```

Listing 29: CSV File Setup with Automatic Naming

Automatic File Management:

- Creates `SimulationData/` directory automatically
- Generates unique filename based on simulation start time
- Example: `reactor_sim_2026-01-31_14-30-45.csv`
- Prevents overwriting previous simulation data

6.7.2 Data Logging at Fixed Intervals

```
def logDataToCSV(self):
    """Log current simulation data to CSV file every 0.5 seconds"""
    if hasattr(self, 'csv_writer') and self.csv_writer:
        time_val = self.duneReactor.t[-1]

        # Only log if 0.5 seconds have passed since last log
        if time_val - self.last_log_time >= self.log_interval:
            neutron_density = self.duneReactor.S[0]
            power_mw = qFuel(self.duneReactor.S[0]) / 1.e6
            reactivity = self.duneReactor.reactivity
            fuel_temp = self.duneReactor.S[7]
            coolant_temp = self.duneReactor.S[8]
            flow_rate = self.duneReactor.mdotC / 1.e3  # g/s to kg/s
            rod_position = self.duneReactor.S[9]

            self.csv_writer.writerow([time_val, neutron_density,
power_mw,
                                      reactivity, fuel_temp,
coolant_temp,
                                      flow_rate, rod_position])
            self.last_log_time = time_val
```

Listing 30: Periodic Data Recording

Logging Strategy: Data is recorded every 0.5 seconds rather than every timestep (0.005s) to:

- Reduce file size by factor of 100
- Maintain adequate time resolution for analysis

- Prevent disk I/O from slowing down simulation
- Enable long-duration simulations without storage issues

6.7.3 CSV File Closure on Exit

```
def closeCSVLogging(self):
    """Close CSV file and flush all buffered data"""
    if hasattr(self, 'csv_file') and self.csv_file:
        self.csv_file.close()
        print(f"CSV file saved: {self.csv_filename}")

def exitSim(self, event):
    """Application exit with cleanup"""
    self.closeCSVLogging()  # Ensure data is saved
    sys.exit()
```

Listing 31: Safe File Handling

Data Integrity: The CSV file is properly closed on application exit, ensuring all buffered data is flushed to disk. Users can analyze simulation results immediately after closing the application.

6.7.4 CSV Data Structure

Each row in the CSV file contains:

Table 3: CSV Data Columns

Column	Units	Description
Time	seconds	Simulation time
Neutron Density	#/cc	Neutron population density
Power	MW	Thermal power output
Reactivity	$	Reactivity in dollars
Fuel Temp	K	Average fuel temperature
Coolant Temp	K	Average coolant temperature
Flow Rate	kg/s	Coolant mass flow rate
Rod Position	%	Control rod withdrawal

Post-Processing Applications:

- Plot reactor transients with Python/MATLAB/Excel
- Calculate integral parameters (energy production, average power)
- Analyze system response to control actions
- Validate PID controller performance
- Generate reports for educational demonstrations

6.8 Arduino Serial Communication

```
def writeToArduino(self):
    """
    Send reactor state to Arduino for physical model control
    Updates servo (rod position), LED (power), and SCRAM indicator
    """
    if self.ser:
        # ===== CONTROL ROD POSITION =====
        # Map rod position (0-100%) to servo angle (5-140 degrees)
        rodWriteOut = abs((self.duneReactor.S[9] / 50.) * 160.)
        if rodWriteOut < 5.0:
            rodWriteOut = 5.0  # Minimum servo position
        elif rodWriteOut > 140.0:
            rodWriteOut = 140.  # Maximum servo position

        # Send command: 'r' + angle
        self.ser.write(("r" + str(int(rodWriteOut))).encode())
        time.sleep(0.1)  # Arduino processing time

        # ===== REACTOR POWER LED =====
        # Map power to LED brightness (0-255)
        maxPwr = 500.  # Maximum power for full brightness [MW]
        normPwr = abs(qFuel(self.duneReactor.S[0]) / 1.e6 / maxPwr)
        normPwr = 250. * normPwr
        if normPwr >= 250:
            normPwr = 250

        # Send command: 'p' + brightness
        self.ser.write(("p" + str(int(normPwr))).encode())
        time.sleep(0.1)

        # ===== COOLANT PUMP CONTROL =====
        # Independent pump speed control based on coolant flow rate
        # Map coolant flow rate (g/s) to motor speed (20-180)
        minFlow = 200.e3  # 200 kg/s = 200000 g/s minimum
        maxFlow = 1200.e3  # 1200 kg/s = 1200000 g/s maximum
        currentFlow = self.duneReactor.mdotC

        # Clamp flow to valid range
        if currentFlow < minFlow:
            currentFlow = minFlow
        elif currentFlow > maxFlow:
            currentFlow = maxFlow

        # Map to motor speed range (20-180)
        motorSpeed = int(20 + (currentFlow - minFlow) / (maxFlow -
   minFlow) * 160)

        # Send command: 'c' + motorSpeed
        self.ser.write(("c" + str(motorSpeed)).encode())
        time.sleep(0.1)

        # ===== SCRAM INDICATOR LED =====
        # Red LED: on during SCRAM, off otherwise
        scramValue = 1 if self.duneReactor.scramToggle else 0

        # Send command: 's' + state
```

```
        self.ser.write(("s" + str(int(scramValue))).encode())
        time.sleep(0.1)
```

Listing 32: Arduino Hardware Feedback Interface

Communication Protocol: The simple character-based protocol ('r', 'p', 'c', 's' commands) provides robust serial communication. Case 'c' enables independent coolant pump control, decoupled from reactor power. Short delays (0.1s) ensure the Arduino has time to process commands and move actuators before the next command arrives.

6.9 Serial Port Auto-Detection

```
def initSerial():
    """
    Attempt to establish serial connection with Arduino
    Tries multiple ports across different platforms
    """
    from sys import platform as _platform
    import serial
    from serial.tools import list_ports
    ser = None

    print("Platform " + _platform + " detected.")
    print("Attempting to establish connection with arduino.")

    # Build list of candidate ports based on platform
    port_candidates = []
    if _platform == "linux" or _platform == "linux2":
        # Linux: /dev/ttyACM* or /dev/ttyUSB*
        port_candidates.extend(["/dev/ttyACM" + str(i) for i in range(10)])
        port_candidates.extend(["/dev/ttyUSB" + str(i) for i in range(10)])
    elif _platform == "windows" or _platform == "win32" or _platform == "win64":
        # Windows: COM ports
        port_candidates.extend(["COM" + str(i) for i in range(1, 11)])
    elif _platform == "darwin":
        # macOS: /dev/cu.usbmodem*
        port_candidates.extend(["/dev/cu.usbmodem14" + str(i + 10)
                                for i in range(10)])

    # Also try any ports detected by pyserial
    port_candidates.extend([p.device for p in list_ports.comports()])

    # Try each port
    tried = set()
    for port in port_candidates:
        if port in tried:
            continue
        tried.add(port)
        try:
            print("Attempting handshake with arduino on " + port + ":9600")
            ser = serial.Serial(port, 9600, timeout=2)
            time.sleep(3)  # Arduino reset on serial connection
            break
```

```
        except Exception as exc:
            print("Connection failed on " + port + " because " + str(exc))
            ser = None

    if not ser:
        print("Arduino Not Detected. Running without serial connection")
    else:
        print("Connection to Arduino Established on " + ser.port)

    return ser
```

Listing 33: Cross-Platform Arduino Connection

Robustness: The port auto-detection tries all possible serial ports on the system, making the software work seamlessly across Linux, Windows, and macOS without user configuration. If no Arduino is found, the simulation continues without hardware feedback.

6.10 Application Entry Point

```
def main():
    """
    Application entry point
    Creates wxPython application and main window
    """
    app = wx.App(False)  # Don't redirect stdout/stderr to GUI
    frame = CalcFrame(None)
    frame.Show(True)
    app.MainLoop()  # Start event loop

if __name__ == "__main__":
    main()
```

Listing 34: Main Application Launcher

7 Arduino Hardware Integration

The Arduino firmware provides physical feedback for the reactor simulation through servo motors, LEDs, and a coolant pump motor integrated with a 3D printed physical reactor model.

7.1 Arduino Sketch Overview

```
#include <Servo.h>

Servo myServo;
int servoPin = 9;           // Control rod servo motor
int ledRGBRedPin = 11;      // SCRAM indicator LED (red)
int ledRGBBluePin = 6;      // Power indicator LED (blue)
int motorPWM = 3;           // Coolant pump motor PWM control

bool scramActive = false;
```

```
10
11 void setup(void){
12   Serial.begin(9600);       // 9600 baud serial communication
13
14   // Initialize servo
15   myServo.attach(servoPin);
16   myServo.write(5);         // Rods fully inserted (default safe state)
17
18   // Initialize LED pins
19   pinMode(ledRGBRedPin, OUTPUT);
20   pinMode(ledRGBBluePin, OUTPUT);
21   analogWrite(ledRGBRedPin, 0);   // Red LED off
22   analogWrite(ledRGBBluePin, 0);  // Blue LED off
23
24   // Initialize motor control
25   pinMode(motorPWM, OUTPUT);
26   analogWrite(motorPWM, 0);       // Motor off initially
27
28   loop();
29 }
```

Listing 35: Arduino Setup and Pin Configuration

Hardware Connections:

- **Servo (Pin 9)**: Positions 3D printed control rod mechanism (5-140° range)
- **Blue LED (Pin 6, PWM)**: Brightness proportional to reactor power
- **Red LED (Pin 11, PWM)**: Illuminates during SCRAM condition
- **Motor (Pin 3, PWM)**: Coolant pump speed varies with power level

7.1.1 Detailed Circuit Implementation with L298N Motor Driver

The Arduino cannot directly drive the 12V pump due to current and voltage limitations. An L298N H-Bridge motor driver module acts as a high-power switch:

Table 4: Complete Circuit Connections

Component	Arduino Pin	Notes
Servo Signal	Pin 9 (Digital)	Control rod position
Blue LED (Anode)	Pin 6 (PWM)	Power indicator
Red LED (Anode)	Pin 11 (PWM)	SCRAM indicator
L298N ENA	Pin 3 (PWM)	Pump speed control
L298N IN1	Logic High	Direction control
L298N IN2	Logic Low	Direction control
L298N OUT-A	Pump (+)	Motor terminal
L298N OUT-A	Pump (-)	Motor terminal
L298N 12V	External PSU (+)	12V power supply
L298N GND	Arduino GND	Common ground

Motor Speed Mapping: The firmware maps normalized power (0-255) to motor PWM (20-180):

- Minimum PWM = 20: Prevents motor stalling at low power
- Maximum PWM = 180: Prevents motor overdriving and noise
- Linear mapping: `motorSpeed = map(ledPowerSetting, 0, 255, 20, 180)`

7.2 Serial Command Processing

```
void loop(void){
  int ledPowerSetting;
  int rodHeightSetting;
  int scramCondition;
  int motorSpeed;
  int coolantFlowSetting;

  if (Serial.available() > 0)
  {
    char inByte = Serial.read();

    switch(inByte)
    {
    case 'p': // Power command
      // Set blue LED brightness (0-255)
      // LED brightness indicates reactor power level
      ledPowerSetting = numberFromSerial();
      analogWrite(ledRGBBluePin, ledPowerSetting);
      break;

    case 'c': // Coolant flow control command
      // Independent pump speed control based on flow rate
      // Receives motor speed value (20-180)
      coolantFlowSetting = numberFromSerial();
      analogWrite(motorPWM, coolantFlowSetting);
      break;

    case 'r': // Rod position command
      rodHeightSetting = numberFromSerial();
      myServo.write(rodHeightSetting);  // Move servo to angle
      break;

    case 's': // SCRAM condition
      scramCondition = numberFromSerial();
      Serial.print("SCRAM received: ");
      Serial.println(scramCondition);

      if (scramCondition > 0) {
        // SCRAM active: red LED on at full brightness
        analogWrite(ledRGBRedPin, 255);
        scramActive = true;
      } else {
        // SCRAM cleared: red LED off
        analogWrite(ledRGBRedPin, 0);
        scramActive = false;
      }
      break;
    }

```

```
    Serial.flush();  // Clear serial buffer
  }
}
```

Listing 36: Arduino Main Loop and Command Parser

Command Protocol Summary:

Table 5: Arduino Serial Commands

Command	Parameter	Hardware	Function
'p'	0-250	Blue LED	Power indicator brightness
'c'	20-180	Pump Motor	Coolant flow rate control
'r'	5-140	Servo	Control rod position
's'	0-1	Red LED	SCRAM alarm indicator

Key Design Change: Case 'c' now provides independent coolant pump control, decoupled from reactor power (case 'p'). This allows:

- Realistic simulation of pump control systems
- Manual flow rate adjustment independent of power level
- Demonstration of cooling system importance
- Investigation of thermal-hydraulic transients

7.2.1 Python-Arduino Communication for Coolant Control

```
# Inside writeToArduino() method:
# send coolant flow control to motor (case 'c')
# Map coolant flow rate (g/s) to motor speed (20-180)
minFlow = 200.e3  # 200 kg/s = 200000 g/s minimum
maxFlow = 1200.e3  # 1200 kg/s = 1200000 g/s maximum
currentFlow = self.duneReactor.mdotC

# Clamp flow to valid range
if currentFlow < minFlow:
    currentFlow = minFlow
elif currentFlow > maxFlow:
    currentFlow = maxFlow

# Map to motor speed range (20-180)
motorSpeed = int(20 + (currentFlow - minFlow) / (maxFlow - minFlow) *
    160)
self.ser.write(("c" + str(motorSpeed)).encode())
time.sleep(0.1)
```

Listing 37: Sending Coolant Flow Commands to Arduino

Flow-to-Speed Mapping:

$$\text{motorSpeed} = 20 + \frac{\dot{m}_{coolant} - \dot{m}_{min}}{\dot{m}_{max} - \dot{m}_{min}} \times 160 \tag{29}$$

where:

- $\dot{m}_{min} = 200$ kg/s (minimum coolant flow)
- $\dot{m}_{max} = 1200$ kg/s (maximum coolant flow)
- Motor PWM range: 20-180 (prevents stalling and overdriving)

7.2.2 Integer Parsing from Serial

```
int numberFromSerial(void)
{
  /*
   * Parse integer from serial buffer
   * Reads digits until end of number or buffer full
   */
  char numberString[8];
  unsigned char index=0;
  delay(10);  // Wait for full number to arrive

  while(Serial.available() > 0)
  {
    delay(10);
    numberString[index++]=Serial.read();
    if(index>6)
    {
      break;  // Limit to 6 digits
    }
  }
  numberString[index]=0;  // Null terminator
  return atoi(numberString);  // Convert string to integer
}
```

Listing 38: Serial Number Parser

Command Protocol Summary:

Table 6: Arduino Serial Commands

Command	Format	Action
Power	'p' + [0-255]	Set blue LED brightness and pump speed
Rod	'r' + [5-140]	Set servo angle (control rod position)
SCRAM	's' + [0 or 1]	Control red LED (0=off, 1=on)

7.3 Physical Model Integration

The 3D printed reactor model controlled by Arduino includes:

1. **Control Rod Mechanism**: Servo-actuated 3D printed structure that raises/lowers to indicate rod position. Provides visual representation of control rod state.
2. **Reactor Core**: Translucent 3D printed housing with blue LED. LED brightness increases with reactor power, simulating Cherenkov radiation glow seen in real reactors.

3. **SCRAM Indicator**: Red LED that illuminates when safety systems activate, providing clear visual feedback of emergency shutdown.

4. **Coolant Pump**: Small DC motor representing coolant circulation. Motor speed increases with power level, demonstrating the need for active cooling at higher powers.

Educational Value: The physical model transforms abstract simulation data into tangible phenomena. Students can see the blue glow intensify as power increases, watch control rods move in response to user commands, and observe the dramatic red SCRAM indicator when safety limits are exceeded. This multi-sensory experience reinforces learning far more effectively than simulation alone.

8 Mathematical Formulation and Numerical Methods

8.1 System of ODEs

The complete reactor system is described by a 10-dimensional system of first-order ODEs:

$$\frac{dn}{dt} = \frac{\rho(t) - \beta}{\Lambda} n + \sum_{i=1}^{6} \lambda_i C_i \tag{30}$$

$$\frac{dC_i}{dt} = \frac{\beta_i}{\Lambda} n \cdot \eta - \lambda_i C_i \quad (i = 1, \ldots, 6) \tag{31}$$

$$\frac{dT_f}{dt} = \frac{Q(n) - hA_c(T_f - T_c)}{m_f C_{p,f}(T_f)} \tag{32}$$

$$\frac{dT_c}{dt} = \frac{hA_c(T_f - T_c) + C_{p,c}(T_{in} - T_c)\dot{m}_c}{m_c C_{p,c}} \tag{33}$$

$$\frac{dh}{dt} = \dot{h}(t) \tag{34}$$

where $\eta = 0.6$ is the neutron lifetime correction factor.

8.2 Reactivity Functions

8.2.1 Control Rod Worth

Differential rod worth (sinusoidal):

$$R(h) = k \sin\left(\frac{\pi h}{100}\right) \tag{35}$$

Integral rod worth:

$$\rho_{rod}(h) = \int_0^h R(h')\, dh' = -\frac{100k}{\pi}\left[\cos\left(\frac{\pi h}{100}\right) - 1\right] \tag{36}$$

8.2.2 Temperature Reactivity

$$\rho_{temp}(T_f) = \alpha_T(T_f - T_{in}) \tag{37}$$

where $\alpha_T = -\frac{0.007 \times 10^{-5}}{\beta}$ [\$/K].

8.3 Numerical Integration Method

8.3.1 LSODA Algorithm

The simulation uses scipy's `odeint`, which implements the LSODA (Livermore Solver for Ordinary Differential Equations with Automatic method switching) algorithm. LSODA automatically detects stiffness and switches between:

- **Non-stiff method**: Adams-Moulton predictor-corrector (up to order 12)
- **Stiff method**: Backward Differentiation Formulas (BDF, up to order 5)

8.3.2 Why LSODA for Reactor Kinetics?

Reactor kinetics exhibits varying stiffness depending on operating conditions:

- **Normal operation**: System is only mildly stiff due to fast prompt neutrons and slow delayed neutrons (time scale separation: 10^{-5} s vs. 10 s).
- **Rapid transients**: System becomes very stiff when reactivity changes quickly (e.g., SCRAM insertion). Prompt neutron population can change on microsecond timescales while precursors evolve over seconds.
- **Temperature coupling**: Thermal dynamics (seconds to minutes) are much slower than neutronics, adding another time scale.

LSODA's adaptive method switching ensures both accuracy and efficiency across all operating regimes.

8.4 Time Stepping Strategy

The simulation uses two time scales:

1. **Physics time step**: $\Delta t_{physics} = 0.005$ s (5 ms)
   - Sufficiently small to resolve control system dynamics
   - Captures reactivity feedback loops
   - Allows smooth PID control response
2. **Display update rate**: $\Delta t_{display} = 1$ s
   - Reduces computational overhead of plot rendering
   - Provides smooth visual updates without flicker
   - Maintains GUI responsiveness

8.5 Stability Analysis: Linearized System

At steady state, the reactor can be approximated by linearizing about an equilibrium point. Consider small perturbations δn, δC_i, δT_f:

$$\frac{d(\delta n)}{dt} \approx \frac{\rho_0 - \beta}{\Lambda}\delta n + \frac{n_0}{\Lambda}\delta\rho + \sum_{i=1}^{6} \lambda_i \delta C_i \tag{38}$$

where $\delta\rho = \alpha_T \delta T_f + R(h_0)\delta h$.
The eigenvalues of the linearized system determine stability:

- **Negative** α_T: Provides negative feedback, stabilizing the system
- **Prompt neutron response**: Fast eigenvalue $\sim (\rho - \beta)/\Lambda$
- **Delayed neutron modes**: Six eigenvalues near $-\lambda_i$ (stable decay modes)

8.6 Power-Temperature Coupling

The coupling between neutronics and thermal-hydraulics creates characteristic response patterns:

8.6.1 Power Increase Scenario

1. Control rods withdrawn $\rightarrow \rho \uparrow$
2. Neutron population increases $\rightarrow n \uparrow$
3. Power increases $\rightarrow Q = Q(n) \uparrow$
4. Fuel temperature rises $\rightarrow T_f \uparrow$
5. Negative temperature feedback $\rightarrow \rho \downarrow$
6. System stabilizes at new equilibrium with higher T_f and higher power

This self-regulating behavior is fundamental to reactor safety. The time constant for this feedback loop is determined by:

$$\tau_{feedback} \approx \frac{m_f C_{p,f}}{hA_c + |n_0 \alpha_T Q'/\rho|} \tag{39}$$

Typical values: $\tau_{feedback} \sim 1 - 10$ seconds for thermal power changes.

9 Usage Examples and Operational Scenarios

9.1 Installation and Setup

9.1.1 Prerequisites

Ensure Python 3.6+ is installed, then install dependencies:

```
cd /path/to/DUNE
python setup.py develop
```

Or using pip:

```
pip install numpy scipy matplotlib wxpython pyserial
```

9.1.2 Launching the Application

After installation, launch the GUI:

```
pyReactor
```

Or run directly:

```
python duneReactor.py
```

9.2 Operational Scenarios

9.2.1 Scenario 1: Reactor Startup from Cold Shutdown

Objective: Bring reactor from zero power to 100 MW using manual rod control.
Procedure:

1. **Initial State**: Rods fully inserted (0%), power = 0 MW, temperatures at 450 K
2. **Action**: Slowly withdraw rods using slider (or type "50" in Rod Setpoint box)
3. **Observation**:
   - Neutron population increases exponentially
   - Power rises on plot
   - Fuel temperature begins increasing
   - Negative temperature feedback reduces reactivity
4. **Stabilization**: System reaches new equilibrium at ∼100 MW
5. **Time scale**: Takes 30-60 seconds for complete stabilization

Key Learning Point: Demonstrates exponential power increase with positive reactivity and natural stabilization via temperature feedback.

9.2.2 Scenario 2: Automatic Power Control

Objective: Use PID controller to maintain steady 200 MW power output.
Procedure:

1. Start reactor at any power level
2. Enter "200" in Power SetPoint box
3. Check "Power Ctrl" checkbox
4. **Observation**:
   - Controller automatically adjusts rod position
   - Power converges to 200 MW
   - Small oscillations may occur (PID tuning dependent)
   - Once stable, power remains constant despite disturbances

Key Learning Point: Demonstrates feedback control systems and importance of automatic regulation in large-scale systems.

9.2.3 Scenario 3: Reactivity-Initiated Accident (RIA)

Objective: Demonstrate SCRAM protection during excessive reactivity insertion.
Procedure:

1. Start at low power (10-20 MW)
2. Rapidly withdraw rods to 100% (type "100" and press Enter)
3. **Observation**:
    - Power increases rapidly (exponential rise)
    - Fuel temperature rises quickly
    - If fuel reaches 1700 K: automatic SCRAM triggers
    - Message: "Fuel Temperature SCRAM setpoint Exceeded"
    - Red SCRAM button activates (and Arduino red LED if connected)
    - Rods instantly insert, power drops rapidly

Key Learning Point: Demonstrates reactor protection systems and inherent safety mechanisms. Shows why SCRAM systems are critical for preventing core damage.
Physics Phenomena Demonstrated:

- **Prompt Jump**: Upon rapid rod withdrawal, power increases instantaneously before settling into a stable period, consistent with the physics of prompt neutron lifetimes ($\sim 10^{-5}$ s).
- **Doppler Defect**: As fuel temperature rises, the rate of power increase slows down due to negative temperature feedback ($\alpha_T < 0$). This validates the inherent safety characteristics of the simulated reactor core.
- **Reactor Period**: Observable exponential power increase with time constant determined by reactivity: $T = \frac{\Lambda}{\rho - \beta}$

9.2.4 Scenario 4: Loss of Coolant Flow

Objective: Investigate thermal response to reduced cooling.
Procedure:

1. Establish steady state at 200 MW
2. Reduce coolant flow: change "1000" to "200" in Coolant Flow Rate box
3. **Observation**:
    - Reduced heat removal causes fuel temperature rise
    - Negative temperature feedback reduces power
    - System stabilizes at lower power, higher temperature
    - Risk of SCRAM if flow too low

Key Learning Point: Illustrates importance of active cooling and thermal-hydraulic coupling in reactor operation.

9.2.5 Scenario 5: Plot Zoom and Time History Analysis

Objective: Examine detailed transient behavior.

Procedure:

1. Perform any transient (rod movement, power change, SCRAM)
2. Adjust "Plot Zoom" slider:
   - Slider left (low %): Zoom in, see recent 5-10 seconds
   - Slider right (high %): Zoom out, see full 100 seconds
3. Analyze temperature and power coupling on dual-axis plot

Key Learning Point: Time-scale analysis reveals fast neutronics (seconds) vs. slow thermal dynamics (tens of seconds).

9.2.6 Scenario 6: Prompt Jump Mode Demonstration

Objective: Observe the characteristic prompt jump phenomenon when reactivity is suddenly inserted.

Procedure:

1. Start reactor at low-to-moderate power level (10-50 MW)
2. **Enable Prompt Jump Mode** by checking the checkbox in the GUI
3. **Observation**:
   - Control rod instantly withdraws by 3%
   - Approximately \$0.003 reactivity is inserted
   - Power increases rapidly (prompt jump)
   - Temperature feedback stabilizes the power
   - Automatic SCRAM protection remains active
4. Uncheck the checkbox to deactivate the mode

Key Learning Point: Demonstrates the rapid power response to positive reactivity insertion. The "prompt jump" occurs because prompt neutrons respond immediately to reactivity changes, while delayed neutrons continue at their previous rate. This shows why reactivity insertions must be carefully controlled in real reactors.

Physics Phenomena Demonstrated:

- **Prompt Response**: Immediate power increase proportional to $\frac{\rho}{\beta-\rho}$ for subcritical insertions
- **Delayed Neutron Effects**: Power stabilizes as delayed neutron population adjusts
- **Temperature Feedback**: Negative α_T provides inherent safety even during rapid transients

9.3 Arduino Hardware Demonstrations

With Arduino connected:

9.3.1 Visual Power Feedback

- Blue LED brightness proportional to reactor power
- Gradually withdraw rods and watch LED intensify
- Mimics Cherenkov radiation in real reactor pools

9.3.2 Control Rod Motion

- Servo moves 3D printed structure indicating rod position
- Students can see mechanical actuation in real-time
- Reinforces connection between GUI slider and physical position

9.3.3 SCRAM Indication

- Red LED illuminates during SCRAM
- Dramatic visual signal of emergency condition
- Clears when SCRAM button pressed again (reactor reset)

9.3.4 Auditory Feedback from Coolant Pump

- 12V DC pump speed varies with reactor power level
- Provides intuitive auditory sense of reactor state
- Users can "hear" the reactor power without looking at screen
- Louder pump noise correlates with higher power output
- Multi-sensory experience enhances learning retention

Measured Performance Metrics:

- Communication latency: ≈ 100 ms between Python and Arduino
- Sufficiently low for educational demonstrations
- Update rate: 500 Hz physics calculations, 1 Hz display updates
- Servo response time: < 200 ms for full range motion

9.4 Suggested Classroom Activities

9.4.1 Activity 1: Find the Critical Rod Position

Goal: Determine rod position for sustained steady state at given power.

Students adjust rods to find position where power stabilizes at target (e.g., 100 MW). Teaches concept of criticality ($k_{eff} = 1$) and equilibrium.

9.4.2 Activity 2: Control System Challenge

Goal: Tune PID parameters for optimal power control.

Advanced students modify PID gains in code to achieve fast response without overshoot. Introduces control theory concepts.

9.4.3 Activity 3: SCRAM Limit Investigation

Goal: Find maximum safe power before SCRAM.

Students increase power setpoint until SCRAM triggers. Learn about thermal limits and protection systems.

9.4.4 Activity 4: Transient Analysis

Goal: Measure reactor time constants.

Students perform step reactivity insertion, measure power doubling time, and compare to theoretical predictions from point kinetics equations.

10 Conclusions and Future Enhancements

10.1 Project Summary

The DUNE project successfully achieves its primary objectives:

1. **Educational Effectiveness**: Provides engaging, interactive introduction to reactor physics for University 1st year students
2. **Scientific Accuracy**: Implements rigorous six-group point kinetics with thermal-hydraulic coupling
3. **Accessibility**: Cross-platform software with intuitive GUI requires no prior coding knowledge
4. **Physical Integration**: Optional Arduino hardware creates tangible connection between simulation and reality
5. **Safety Education**: SCRAM protection system demonstrates reactor safety principles in controlled environment

10.2 Key Accomplishments

10.2.1 Technical Achievements

- Real-time ODE solution with adaptive stiff/non-stiff integration
- Dual control modes (manual and automatic PID)
- Flow-dependent heat transfer with temperature-dependent properties
- Realistic control rod worth functions based on neutron importance
- Xenon-135 and Samarium-149 fission product poisoning with complete decay chains

- Fuel burnup and isotope depletion tracking (U-235, U-238, Pu-239, fission products)
- 20-dimensional state vector capturing all reactor physics phenomena
- Eight-panel real-time visualization with scrollable value display panel
- Comprehensive CSV data logging for post-simulation analysis
- Stable numerical methods preventing unphysical solutions

10.2.2 Educational Impact

- Visual feedback reinforces abstract concepts (power, temperature, control)
- Interactive experimentation encourages active learning
- Safe environment for exploring reactor accidents and safety systems
- Preparation for university-level nuclear engineering studies

10.3 Limitations of Current Implementation

10.3.1 Physics Simplifications

1. **Point Kinetics Approximation**: Assumes spatial flux shape remains constant (no spatial effects)
    - Cannot model xenon oscillations (spatial flux tilting)
    - Cannot simulate azimuthal or radial power distribution changes
    - Valid for control system analysis and global power transients
    - Adequate for educational demonstrations of core-average behavior
2. **Single Fuel/Coolant Temperature**: Neglects axial and radial temperature distributions
    - Lumped parameter model treats entire core as single node
    - Real reactors have significant temperature gradients
    - Cannot predict hot spots or local heat flux peaking
3. **Simplified Heat Transfer**: Constant or simple flow-dependent h; real correlations more complex
    - Uses simplified Dittus-Boelter-like correlation: $h \propto \dot{m}^{0.8}$
    - Real systems require geometry-specific correlations
    - Neglects effects of boiling and two-phase flow
4. **Xenon-135 and Samarium-149 Poisoning Implemented**: Full decay chain dynamics
    - Xe-135: Models production from fission and I-135 decay, Xe-135 decay, and neutron burnout

- Sm-149: Models complete Nd-149 → Pm-149 → Sm-149 decay chain
- Demonstrates load-following challenges and poison transients
- Captures xenon pit phenomenon during shutdown
- Shows long-term samarium equilibrium buildup
- *Note:* Spatial poison oscillations require multi-dimensional kinetics

5. **Burnup and Isotope Depletion Implemented**: Tracks fuel evolution over time
   - U-235 depletion through fission ($\sigma_f = 585$ barns)
   - U-238 transmutation through neutron capture ($\sigma_c = 2.68$ barns)
   - Pu-239 production from U-238 and consumption through fission ($\sigma_f = 750$ barns)
   - Fission product accumulation from both U-235 and Pu-239 fission
   - Burnup tracking in MWd/kgU units
   - Burnup reactivity coefficients scaled for demonstration timescales
   - GUI displays: Burnup, U-235, U-238, Pu-239 concentrations in real-time
   - CSV logging includes all isotope concentrations
   - *Note:* Accelerated timescale for demonstration (real burnup occurs over months/years)
6. **Excess Reactivity Model**: Fresh core starts subcritical with rods inserted
   - Fresh fuel has \$0.05 excess reactivity built in
   - Rods must be withdrawn to achieve criticality
   - Control rod worth: \$0.2 (0% to 100% withdrawal)
   - Realistic startup behavior requiring rod manipulation
7. **Serial Communication Latency**: Approximately 100 ms delay between simulation and hardware response
   - Caused by timer update rates and serial buffer processing
   - Acceptable for educational purposes
   - Could be reduced with optimized communication protocol

10.3.2 Control System Limitations

1. **Fixed PID Gains**: No adaptive tuning or gain scheduling
2. **No Anti-Windup**: Integral term can accumulate during saturation
3. **Single Control Bank**: Real reactors have multiple independent control rod banks

10.4 Future Enhancement Opportunities

10.4.1 Physics Improvements

1. **Additional Fission Product Poisons**: Extend beyond Xe-135 and Sm-149
   - Pm-149 transient effects (currently modeled only as Sm-149 precursor)
   - Other significant absorbers (Rh-103, Cd isotopes)
2. **Spatial Xenon Oscillations**: Extend to multi-dimensional kinetics
   - Demonstrate spatial power tilting in large cores
   - Regional xenon oscillation control strategies
3. **Multi-Region Model**: Implement core, reflector, and pressure vessel regions
   - More accurate neutron leakage
   - Region-dependent temperature effects
4. **Improved Thermal-Hydraulics**:
   - Axial temperature profiles
   - Two-phase flow modeling (boiling)
   - DNB (Departure from Nucleate Boiling) limits
5. **Fuel Performance Models**:
   - Fission gas release
   - Pellet-cladding interaction
   - Fuel swelling and densification

10.4.2 Software Enhancements

1. **Advanced Control Algorithms**:
   - Model Predictive Control (MPC)
   - Fuzzy logic control
   - Optimal control with state estimation
2. **3D Visualization**:
   - OpenGL reactor core rendering
   - Animated neutron flux distribution
   - Temperature heatmaps
3. **Data Logging and Analysis**:
   - Export to CSV/HDF5 for external analysis
   - Statistical analysis tools
   - Comparison with historical data

4. **Multiplayer Mode**:
    - Networked control stations
    - Team-based reactor operation scenarios
    - Competitive challenges

10.4.3 Hardware Expansions

1. **Enhanced Arduino Features**:
    - Rotary encoder for analog rod control
    - LCD display showing key parameters
    - Alarm buzzer for SCRAM events
    - Temperature sensors for ambient monitoring
2. **Larger Physical Models**:
    - Multiple control rod banks
    - Pressurizer model with pressure control
    - Steam generator visualization
    - Turbine generator integration
3. **Virtual/Augmented Reality**:
    - VR reactor control room
    - AR overlays on physical LEGO model
    - Immersive training environment

10.4.4 Educational Curriculum Development

1. **Guided Tutorials**:
    - Step-by-step walkthroughs for beginners
    - Progressive difficulty levels
    - Interactive quizzes and assessments
2. **Scenario Library**:
    - Historical reactor incidents (TMI, Chernobyl analysis)
    - Advanced reactor concepts (MSR, SMR demonstrations)
    - Competition scenarios for student teams
3. **Integration with LMS**:
    - Canvas/Moodle integration
    - Automated grading of reactor operations
    - Progress tracking and analytics

10.5 Broader Impact

10.5.1 STEM Education

The DUNE project demonstrates the power of simulation-based learning in STEM education. By making complex physics accessible through visualization and interaction, it lowers barriers to understanding advanced topics and inspires students to pursue careers in science and engineering.

10.5.2 Cyber-Physical Systems in Engineering Education

The integration of computation with physical processes—known as Cyber-Physical Systems (CPS)—has proven highly effective in engineering education:

- **SCADA System Analog**: DUNE mimics real-world Supervisory Control and Data Acquisition systems where digital signals drive physical actuators
- **Multi-Sensory Learning**: Combines visual (LED brightness), auditory (pump noise), and tactile (servo motion) feedback
- **Real-Time Constraints**: Students experience the challenges of real-time control and hardware interfacing
- **Cross-Domain Integration**: Bridges nuclear physics, control theory, embedded systems, and software engineering

10.5.3 Public Understanding of Nuclear Energy

Beyond classroom use, DUNE serves as a tool for public outreach and science communication. Interactive demonstrations at science fairs and museums help demystify nuclear technology and inform evidence-based public discourse on energy policy.

10.5.4 Open-Source Community

As an open-source project under the MIT license, DUNE enables:

- Free access for educators worldwide
- Community contributions and improvements
- Adaptation to local educational needs
- Foundation for derivative educational tools

10.6 Project Heritage and Attribution

The DUNE project builds upon the foundational work of the pyReactor educational simulator:

- **Original pyReactor**: Developed by W. Gurecky as an open-source reactor kinetics educational tool
- **Repository**: https://github.com/wgurecky/pyReactor

- **DUNE Enhancement**: Extended for University of Dhaka's Nuclear Engineering program with improved hardware integration using 3D printed components, enhanced GUI features, and comprehensive documentation
- **DUNE Repository**: https://github.com/Hridoy-Kabiraj/DUNE
- **Course Integration**: Developed as part of NE-3206 curriculum at the Department of Nuclear Engineering, University of Dhaka

This collaborative evolution exemplifies the power of open-source software in advancing university-level nuclear engineering education.

10.7 Final Remarks

The DUNE project represents a successful fusion of rigorous physics modeling, software engineering, and educational pedagogy. Its combination of accurate simulation, intuitive interface, and physical hardware feedback creates a uniquely effective learning experience. As nuclear energy plays an increasingly important role in addressing climate change, tools like DUNE will be essential for educating the next generation of nuclear engineers and informed citizens.

The project's modular architecture and open-source nature ensure it will continue evolving, incorporating new features and adapting to emerging educational needs. Whether used in an undergraduate nuclear engineering course, university reactor theory laboratory, or public science museum, DUNE serves its core mission: making nuclear reactor physics accessible, understandable, and inspiring.

"The reactor is not just a source of energy, but a testament to human ingenuity in harnessing the fundamental forces of nature."

Acknowledgements

We would like to express our sincere gratitude to:

Saad Islam
Lecturer
Department of Nuclear Engineering
University of Dhaka

For his invaluable guidance, support, and supervision throughout this project. His expertise in nuclear reactor physics and dedication to student learning made this educational tool possible.

We also acknowledge:

- **W. Gurecky**: For developing the original pyReactor simulator that served as the foundation for this enhanced DUNE implementation
- **Open-Source Community**: For developing the excellent Python scientific computing ecosystem (NumPy, SciPy, Matplotlib, wxPython)

- **Arduino Community**: For providing accessible hardware platforms and comprehensive documentation
- **Department of Nuclear Engineering, University of Dhaka**: For providing the resources and environment conducive to this research

The DUNE project demonstrates how collaborative open-source development can create powerful educational tools for university-level nuclear engineering education.

A Reactor Physics Equations Reference

A.1 Point Kinetics Equations (Standard Form)

For a reactor with I delayed neutron groups:

$$\frac{dn(t)}{dt} = \frac{\rho(t) - \beta}{\Lambda} n(t) + \sum_{i=1}^{I} \lambda_i C_i(t) \tag{40}$$

$$\frac{dC_i(t)}{dt} = \frac{\beta_i}{\Lambda} n(t) - \lambda_i C_i(t) \quad (i = 1, \ldots, I) \tag{41}$$

A.2 Reactor Period

For constant reactivity ρ, the asymptotic reactor period T is:

$$T = \frac{\Lambda}{\rho - \beta} \quad (\text{for } \rho < \beta) \tag{42}$$

A.3 Prompt Jump Approximation

For step reactivity insertion $\Delta\rho$ at $t = 0$:

$$\frac{n(t = 0^+)}{n(t = 0^-)} \approx \frac{\rho_0 + \Delta\rho}{\rho_0} \tag{43}$$

B Thermodynamic Properties

B.1 UO_2 Fuel Properties

- Density: $\rho_{UO_2} = 10.97$ g/cm^3 (theoretical), 10.5 g/cm^3 (typical sintered)
- Specific heat: $C_p = 245 + 0.0586(T - 273)$ J/kg·K
- Thermal conductivity: $k = \frac{1}{7.5408\times10^{-3}+1.7692\times10^{-5}T+3.6142\times10^{-9}T^2}$ W/m·K
- Melting point: 3120 K (2847°C)

B.2 Water/Steam Properties (at 15 MPa)

- Saturation temperature: 615 K (342°C)
- Liquid density: 700 kg/m^3
- Liquid specific heat: $C_{p,l} = 4.18$ kJ/kg·K
- Latent heat of vaporization: 931 kJ/kg

References

[1] Lamarsh, J. R., & Baratta, A. J. (2001). *Introduction to Nuclear Engineering* (3rd ed.). Prentice Hall.

[2] Duderstadt, J. J., & Hamilton, L. J. (1976). *Nuclear Reactor Analysis*. Wiley.

[3] Hetrick, D. L. (1993). *Dynamics of Nuclear Reactors*. American Nuclear Society.

[4] Keepin, G. R. (1965). *Physics of Nuclear Kinetics*. Addison-Wesley.

[5] OECD/NEA (2008). *Delayed Neutron Data for the Major Actinides*. NEA Data Bank.

[6] Virtanen, P., et al. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17, 261–272.

[7] Hunter, J. D. (2007). Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3), 90-95.

[8] Gurecky, W. (2025). *pyReactor: Educational Nuclear Reactor Simulator*. GitHub repository. https://github.com/wgurecky/pyReactor

[9] Waltar, A. E., & Reynolds, A. B. (1981). *Fast Breeder Reactors*. Pergamon Press.

[10] Arduino LLC. (2025). *Arduino Uno R3 Datasheet*. Available: https://www.arduino.cc

[11] Python Software Foundation. (2025). *The Python Standard Library*. Available: https://docs.python.org/3/library/