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

January 27, 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 and control rod dynamics, offering real-time visualization of reactor behavior including power transients, temperature evolution, 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

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

- Coupled thermal-hydraulic modeling with fuel and coolant temperature tracking

- Interactive 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)

- Live plotting of reactor power, fuel temperature, and coolant temperature

- Optional Arduino integration for physical model control (servo motors, LED feedback)

- Temperature-dependent reactivity feedback modeling

- Configurable coolant flow rate control

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
 legoReactor.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
 pyReactor.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**: `LegoReactor`

- **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   legoReactor.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**: pyReactor

- **Entry Point**: `pyReactor` 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   Complete State Vector

The complete reactor system is described by a 10-dimensional state vector:

$$\mathbf{S}(t) = [n(t), C_1(t), C_2(t), C_3(t), C_4(t), C_5(t), C_6(t), T_{fuel}(t), T_{coolant}(t), h(t)]^T \tag{12}$$

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

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

```python
#!/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%)
```

```
22 # This is the key parameter that makes reactor control possible
23 # Without delayed neutrons, reactors would be uncontrollable
24
25 Lamb = 10.e-5
26 # Average neutron lifetime (generation time) [s]
27 # For thermal reactors: ~10^-5 seconds
28 # 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.

```
1  v = 2200.e3
2  # Neutron velocity cm/s
3  # This is the thermal neutron velocity at 20 C
4  # Corresponds to 0.025 eV neutron energy (thermal equilibrium)
5
6  Ef = 3.204e-11
7  # Energy per fission [J]
8  # For U-235: approximately 200 MeV per fission
9  # Converted to Joules: 200 MeV * 1.602e-13 J/MeV
10
11 Sigma_f = 0.0065
12 # Macrosopic fission cross section in reactor [1/cm]
13 # This depends on fuel enrichment and geometry
14 # Represents probability of fission per unit path length
15
16 Vr = 3.e6
17 # Reactor volumue [cc] = 3 cubic meters
18 # Typical small research reactor size
19 # Determines total power capacity and heat generation
20
21 Lc = Lamb * v
22 # Mean neutron travel length in core [cm]
23 # = (10^-5 s) * (2.2    10^6 cm/s) = 22 cm
24 # Characteristic diffusion length scale
25
26 VfFuel = 0.4
27 # Volume fraction occupied by fuel
28 # Remaining 60% is coolant, structure, etc.
29
30 VfH2O = 1. - VfFuel
31 # 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.

```
1 hc = 1.
2 # W/cm^2 * K avg heat transfer coeff between fuel and water
3 # This is a simplified constant coefficient
4 # In reality, h depends on flow rate, temperature, geometry
```

```
5  # Typical values: 0.5-2 W/cm^2 K  for LWR conditions
6
7  Ac = 4.e5
8  # cm^2  fuel to coolant contact area
9  # Large surface area needed for efficient heat removal
10 # = 40 m^2 for 3 m^3 reactor (reasonable geometric estimate)
11
12 Tin = 450.
13 # K coolant inlet temperature
14 # = 177 C , typical for pressurized water reactors
15 # Below boiling point at atmospheric pressure
```

Listing 3: Heat Transfer Parameters

4.2 Reactivity Feedback Coefficient

```
1  alphaT = -0.007 * 1.e-5 / beta
2  # pcm / K / beta  reactivity per kelvin
3  # Negative temperature coefficient provides stability
4  # As fuel heats up, reactivity decreases (negative feedback)
5  # Magnitude: -0.007 pcm/K / 0.0065     -0.0011 $/K
6  # This self-regulating behavior is crucial for reactor safety
7
8  # Number of delayed neutron groups
9  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

```
1  def dndt(S, t, reactivity):
2      """
3      Time derivative of neutron population with 6 delayed neutron groups
        .
4
5      Implements: dn/dt = [(rho - beta)/Lambda] * n + sum(lambda_i * C_i)
6
7      Args:
8          S: State vector [n, C1, C2, C3, C4, C5, C6, Tfuel, Tcoolant, h]
9          t: Time (not explicitly used, but required by ODE solver)
10         reactivity: Current reactivity in dollars
11
12     Returns:
13         dn/dt: Rate of change of neutron density [neutrons/cm^3/s]
14     """
15     # Sum contributions from all delayed neutron groups
16     # Each group releases neutrons at rate lambda_i * C_i
17     delayed_contribution = np.sum(lambda_i * S[1:7])
18
19     # Main point kinetics equation
20     # Prompt neutron contribution: (reactivity - beta) / Lambda * n
```

```
21     # Delayed neutron contribution: sum of all precursor decay rates
22     ndot = (reactivity - beta) / Lamb * S[0] + delayed_contribution
23
24     # Prevent unphysical negative neutron population
25     # If neutrons are already zero and decreasing, set derivative to
   zero
26     if S[0] <= 0. and ndot < 0.:
27         return 0.
28     else:
29         return ndot
```

<div align="center">Listing 5: Neutron Density Time Derivative</div>

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

4.4 Precursor Population Dynamics

```
1  def dCdt(S, t, group_index):
2      """
3      Time derivative of delayed neutron precursor population for a
   specific group.
4
5      Implements: dC_i/dt = (beta_i/Lambda) * n - lambda_i * C_i
6
7      Args:
8          S: State vector
9          t: Time
10         group_index: Which precursor group (0-5 for groups 1-6)
11
12     Returns:
13         dC_i/dt: Rate of change of precursor concentration [nuclei/cm
   ^3/s]
14     """
15     lifetimeCorrection = 0.6
16     # Empirical correction factor accounting for 4-factor formula
17     # Compensates for neutron losses during moderation and diffusion
18     # Effective survival probability from birth to next fission
19
20     C_index = group_index + 1
21     # Precursor concentration stored at S[1] to S[6]
22     # Index 0 is neutron density
23
24     # Production of precursors from fissions
25     # beta_i fraction of neutrons create this precursor type
26     production = (beta_i[group_index] / Lamb) * S[0] *
   lifetimeCorrection
27
28     # Loss of precursors due to radioactive decay
29     # Releases delayed neutrons at rate lambda_i
30     decay = lambda_i[group_index] * S[C_index]
31
32     Cdot = production - decay
33
34     # Prevent unphysical negative precursor concentrations
```

```
35     if S[C_index] < 0. and Cdot < 0.:
36         return 0.
37     else:
38         return Cdot
```

Listing 6: Delayed Neutron Precursor Evolution

Implementation Note: The `lifetimeCorrection` factor (0.6) accounts for neutron losses not explicitly modeled in the point kinetics approximation, such as leakage and parasitic absorption. This ensures the steady-state precursor concentrations match the expected equilibrium values.

4.5   Thermal Power Calculation

```
1  def qFuel(n):
2      """
3      Given neutron population return thermal power
4
5      Power [W] = Volume * fuel_fraction * (neutron_density * velocity)
6                  * fission_cross_section * energy_per_fission
7
8      Args:
9          n: Neutron density [neutrons/cm^3]
10
11     Returns:
12         Thermal power [W]
13     """
14     return Vr * VfFuel * (n * v) * Sigma_f * Ef
```

Listing 7: Fission Power Calculation

Physical Interpretation: This calculates power from neutron density using:

$$Q = V_r \cdot V_{f,fuel} \cdot (n \cdot v) \cdot \Sigma_f \cdot E_f \tag{13}$$

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

```
1  def dTfdt(S, t, mdotC):
2      """
3      Time derivative of fuel temperature with improved heat transfer.
4      Uses temperature-dependent properties and Dittus-Boelter
       correlation.
5
6      Energy balance: Heat in (fission) = Heat out (to coolant) + Heat
       stored
7
8      Args:
9          S: State vector
10         t: Time
11         mdotC: Coolant mass flow rate [g/s]
12
13     Returns:
14         dT_fuel/dt [K/s]
```

```
15      """
16      # Temperature-dependent UO2 heat capacity (J/g*K)
17      T_fuel = S[7]  # fuel temperature at index 7
18      CpUO2 = 0.2455 + 5.86e-5 * (T_fuel - 273.15)
19      # Linear temperature dependence for UO2
20      # Base value ~245 J/kg K at room temp
21      # Increases slightly with temperature
22
23      densityUO2 = 12.5  # g/cc (theoretical density ~10.97, using higher
        for sintered fuel)
24
25      # Improved heat transfer coefficient using flow-dependent
        correlation
26      # Dittus-Boelter-like correlation: h = h0 * (mdot/mdot0)^0.8
27      h0 = 1.5  # W/cm^2*K baseline heat transfer coefficient
28      mdot0 = 1000.e3  # reference flow rate [g/s] = 1000 kg/s
29      h = h0 * (mdotC / mdot0) ** 0.8
30      # Heat transfer improves with flow rate (turbulent convection)
31      # Power 0.8 from Dittus-Boelter correlation for turbulent flow
32
33      # Heat generation from fission
34      heat_in = qFuel(S[0])
35
36      # Heat removal to coolant
37      heat_out = Ac * h * (S[7] - S[8])  # Newton's law of cooling
38
39      # Fuel thermal mass
40      fuel_mass = densityUO2 * VfFuel * Vr
41
42      # Energy balance: dE/dt = Q_in - Q_out
43      # dE/dt = m * Cp * dT/dt
44      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

```
1  def dTcdt(S, t, mdotC):
2      """
3      Time derivative of water coolant with improved heat transfer.
4      Uses temperature-dependent properties.
5
6      Energy balance for coolant:
7      Heat in (from fuel) + Heat in (from inlet flow) = Heat out (exit
       flow) + Heat stored
8
9      Args:
10         S: State vector
11         t: Time
12         mdotC: Coolant mass flow rate [g/s]
13
14     Returns:
```

```python
15          dT_coolant/dt [K/s]
16      """
17      # Temperature-dependent water properties
18      T_coolant = S[8]   # coolant temperature at index 8
19      CpH2O = 4.2 - 0.0005 * (T_coolant - 273.15)
20      # Water heat capacity in J/g*K
21      # Decreases slightly with temperature (accurate for liquid water)
22      # At 25 C : ~4.18 J/g K
23
24      densityH2O = 1.0  # g/cc (simplified, actual density varies with T)
25
26      # Use same improved heat transfer coefficient
27      h0 = 1.5
28      mdot0 = 1000.e3
29      h = h0 * (mdotC / mdot0) ** 0.8
30
31      # Heat transferred from fuel to coolant
32      heat_from_fuel = Ac * h * (S[7] - S[8])
33
34      # Energy brought in by inlet coolant flow
35      # mdotC * Cp * (Tin - Tcoolant)
36      # If coolant is hotter than inlet, this term is negative (cooling
    effect)
37      heat_from_inlet = CpH2O * (Tin - S[8]) * mdotC
38
39      # Coolant thermal mass
40      coolant_mass = densityH2O * Vr * CpH2O
41
42      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
1 def diffRodWorth(h):
2     """
3     Improved differential control rod worth curve using cosine shape.
4     Tuned to keep total worth ~0.5 $ so small height moves are gentler.
5     h is fractional height: h=0 is fully inserted, h=100 is fully
    withdrawn
6     delta_h * R(h) = reactivity change
7
8     Physical basis: Rod worth follows importance-weighted distribution
9     Maximum near core midplane (h=50), zero at top/bottom
10
11    Args:
12        h: Rod height position [%] (0 = inserted, 100 = withdrawn)
13
14    Returns:
15        Differential rod worth [$/% height]
16    """
17    scalingFac = 0.005 * 1.e-5 / beta
```

```python
18      # Tuning factor to achieve desired total rod worth
19      # Result: total integral worth    0.2-0.5 dollars
20
21      # Sinusoidal shape: peaks at h=50 (core midplane)
22      return scalingFac * np.sin(np.pi * h / 100.0) * 100.0
23
24  def intRodWorth(h1, h2):
25      """
26      Integral control rod worth curve.
27      Returns reactivity in dollars ($/Beta)
28
29      Integrates differential worth from position h1 to h2
30      rho = integral from h1 to h2 of R(h) dh
31
32      Args:
33          h1: Initial rod position [%]
34          h2: Final rod position [%]
35
36      Returns:
37          Reactivity change [dollars]
38      """
39      scalingFac = 0.005 * 1.e-5 / beta
40
41      # Analytical integral of sin(pi*h/100):
42      # integral = -100/pi * cos(pi*h/100)
43      integral = lambda h: -100.0 * scalingFac * (100.0 / np.pi) * np.cos
    (np.pi * h / 100.0)
44
45      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
1  def rho(S, t, hrate, deltaT):
2      """
3      Temperature and control rod reactivity.
4      Reactivity in units of Dollars  (deltaK / Beta)
5      Takes control rod movement rate in (%/s)
6
7      Total reactivity = control rod component + temperature feedback
8
9      Args:
10         S: State vector
11         t: Time
12         hrate: Rod movement rate [%/s] (not used in calculation, but
    kept for interface)
13         deltaT: Time step (not used, but kept for interface)
14
15     Returns:
16         Total reactivity [dollars]
17     """
```

```
18      # Temperature feedback: negative for stability
19      # Alpha_T * (T_fuel - T_inlet)
20      temp_reactivity = alphaT * (S[7] - Tin)
21
22      # Control rod reactivity: integral worth from 0 to current position
23      # More withdrawn (higher h) = more positive reactivity
24      rod_reactivity = intRodWorth(0., S[9])
25
26      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

```
1  def reactorSystem(S, t, hrate, deltaT, mdotC=1000.e3):
2      """
3      Complete reactor system with 6 delayed neutron groups.
4      State vector S = [n, C1, C2, C3, C4, C5, C6, Tfuel, Tcoolant,
   rodPosition]
5
6      This function returns dS/dt for the entire 10-dimensional state
   vector.
7      Called by scipy's ODE integrator (odeint).
8
9      Args:
10         S: Current state vector (length 10)
11         t: Current time [s]
12         hrate: Control rod movement rate [%/s]
13         deltaT: Time step (for reference, not used in continuous ODEs)
14         mdotC: Coolant mass flow rate [g/s]
15
16     Returns:
17         dS/dt: Time derivatives of all state variables
18     """
19     # Calculate current reactivity based on state
20     reactivity = rho(S, t, hrate, deltaT)
21
22     # Build derivative vector
23     # Start with neutron population derivative
24     dSdt = [dndt(S, t, reactivity)]
25
26     # Add all 6 precursor group derivatives
27     for i in range(NUM_GROUPS):
28         dSdt.append(dCdt(S, t, i))
29
30     # Add temperature derivatives (fuel and coolant)
31     dSdt.append(dTfdt(S, t, mdotC))
32     dSdt.append(dTcdt(S, t, mdotC))
33
34     # Add control rod position derivative
35     # Rod position changes at specified rate
36     dSdt.append(hrate)
```

```
37
38      return dSdt
```

<div align="center">Listing 12: Full Reactor System Differential Equations</div>

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   LegoReactor Class Structure

```python
1  class LegoReactor(object):
2      """
3      Provides methods to interact with the point kenetics model.
4      The reactor system state vector (with 6 delayed neutron groups):
5      S = [neutrons/cc, C1, C2, C3, C4, C5, C6, fuelT, coolantT,
   rodPosition]
6      """
7      def __init__(self, initialSystemState=None, tstep=0.01):
8          """ Initialize reactor system state """
9          if initialSystemState is None:
10             # Default initial conditions
11             n0 = 5.e7  # Initial neutron density [neutrons/cm^3]
12             # Corresponds to ~1 MW thermal power at steady state
13
14             # Initialize precursor concentrations
15             # At equilibrium: C_i = (beta_i / lambda_i / Lambda) * n
16             # Simplified here: distribute total beta equally among
   groups
17             C_init = [n0 * 0.0065 / 6.0] * 6
18
19             # Complete initial state:
20             # [n, C1-C6, Tfuel, Tcoolant, rodPosition]
21             initialSystemState = [n0] + C_init + [450., 450., 0.]
22             # Tfuel = Tcoolant = 450 K (isothermal start)
23             # Rod position = 0% (fully inserted, subcritical)
24
25         self.S = np.array(initialSystemState)
26         self.reactivity = rho(self.S, 0, 0, 0)
27         self.tstep = tstep  # Time step for integration [s]
28         self.t = np.array([0, self.tstep])
29
30         # Control variables
31         self.hrate = 0.0  # Rod movement rate [% / s]
32         self.rodSetPoint = 0.0  # Desired rod position [%]
33         self.mdotC = 1000.e3  # Coolant flow rate [g / s] = 1000 kg/s
34         self.coolantSetPoint = 1000.e3
35         self.pwrCtrl = False  # Power control mode off by default
36         self.scramToggle = False  # SCRAM status
37
```

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

Listing 13: LegoReactor 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

```
1  def timeStep(self):
2      """ Step reactor system forward in time """
3      # Pre-step checks: control logic, safety systems
4      self.__preStep()
5
6      # Integrate ODEs from t to t+tstep
7      # odeint returns array of solutions; take the last (final) value
8      self.S = integrate.odeint(reactorSystem, self.S, self.t,
9                                args=(self.hrate, self.tstep, self.mdotC)
      )[-1]
10
11     # Update reactivity based on new state
12     self.reactivity = rho(self.S, 0, 0, 0)
13
14     # Advance time
15     self.t += self.tstep
16
17     # Update rolling data storage
18     # Shift arrays left (discard oldest), append new values
19     self.storVals = np.roll(self.storVals, -1, axis=1)
20     self.time = np.roll(self.time, -1)
21     self.time[-1] = self.t[-1]
22
23     # Store key values for plotting
24     self.storVals[:, -1] = np.array([
25         self.S[0],              # Neutron density
26         np.sum(self.S[1:7]),    # Total precursor concentration
27         self.S[7],              # Fuel temperature
28         self.S[8],              # Coolant temperature
29         self.S[9]               # Rod position
30     ])
```

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

```python
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

```python
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
```

```
22
23 def SCRAM(self, scramToggle=True):
24     """
25     You crashed the reactor.
26     Initiate emergency shutdown by rapid rod insertion.
27     """
28     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

```
1 def __rodCtrl(self):
2     """
3     Manual control rod positioning mode
4     Smoothly moves rods toward setpoint using tanh function
5     """
6     # Calculate position error
7     diff = self.S[9] - self.rodSetPoint
8
9     # Apply tanh function for smooth approach
10    # tanh provides fast initial movement that slows near target
11    fnDiff = np.tanh(1.0 * abs(diff))
12
13    if diff < 0.:
14        # Current position below setpoint: withdraw rods (increase h)
15        self.hrate = 0.5 * fnDiff  # Maximum 0.5%/s withdrawal
16    elif diff > 0.:
17        # Current position above setpoint: insert rods (decrease h)
18        self.hrate = -0.5 * fnDiff  # Maximum 0.5%/s insertion
19    else:
20        # At setpoint: no movement
21        self.hrate = 0.
22
23 def setRodPosition(self, rodPos):
24     """Set desired rod position [%]"""
25     self.rodSetPoint = rodPos
26
27 def setRodRate(self, rodRate):
28     """Directly set rod movement rate [%/s] (manual override)"""
29     if not self.pwrCtrl:
30         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

```python
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

```python
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 19: Coolant Flow Rate Control with Ramping

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.

5.8 Testing and Validation

```python
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
    legoReactor = LegoReactor()

    # ===== TEST 1: Manual Rod Control =====
    legoReactor.setRodPosition(50.)  # Withdraw rods to 50%
```

```
12    while i < 10000:
13        legoReactor.timeStep()
14        print("===================================")
15        print("Time [s] = %f" % legoReactor.t[-1])
16        print("Rod percent Withdrawn = %f" % legoReactor.S[9])
17        print("Reactor Power [MW] = %f " % float(qFuel(legoReactor.S
    [0]) / 1.e6))
18        print("Tfuel [K] = %f ,  Tcoolant [K] = %f" % (legoReactor.S
    [7], legoReactor.S[8]))
19        i += 1
20
21    # ===== TEST 2: Automatic Power Control =====
22    i = 0
23    legoReactor.togglePwrCtrl(200.)  # Set target power to 200 MW
24    while i < 10000:
25        legoReactor.timeStep()
26        print("===================================")
27        print("Time [s] = %f" % legoReactor.t[-1])
28        print("Rod percent Withdrawn = %f" % legoReactor.S[9])
29        print("Reactor Power [MW] = %f " % float(qFuel(legoReactor.S
    [0]) / 1.e6))
30        print("Tfuel [K] = %f ,  Tcoolant [K] = %f" % (legoReactor.S
    [7], legoReactor.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 20: Reactor Test Function

6   GUI Frontend: legoReactor.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

```python
class CalcFrame(gui.MyFrame1):
    """
    Main application frame inheriting from auto-generated GUI template
    Implements all event handlers and control logic
    """
    def __init__(self, parent):
        # Initialize parent class (GUI layout)
        gui.MyFrame1.__init__(self, parent)

        # Set initial conditions
        self.setInitConds()

        # Initialize Arduino serial connection (if available)
        self.ser = initSerial()

        # Create reactor simulation instance
        self.legoReactor = rct.LegoReactor(tstep=0.005)
        # Small time step (5 ms) for smooth real-time operation

        # Generate initial data
        self.legoReactor.timeStep()
        self.data = [self.legoReactor.time, self.legoReactor.storVals]

        # Initialize coolant flow display
        self.coolantBox.SetValue(str(round(self.legoReactor.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 21: 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

```python
def init_plot(self):
    """
    Initialize matplotlib figure and axes
```

```python
 4      Creates two subplots: power trace and temperature profiles
 5      """
 6      self.dpi = 100
 7      self.fig = Figure((6.0, 7.0), dpi=self.dpi)
 8
 9      # Subplot 1: Reactor power [MW]
10      self.axes1 = self.fig.add_subplot(211)
11
12      # Subplot 2: Temperatures (dual y-axis)
13      self.axes2 = self.fig.add_subplot(212)
14      self.axes3 = self.axes2.twinx()  # Secondary y-axis for coolant
     temp
15
16      # Set background color
17      if LooseVersion(matplotlib.__version__) >= LooseVersion('2.0.0'):
18          self.axes1.set_facecolor('white')
19      else:
20          self.axes1.set_axis_bgcolor('white')
21
22      self.axes1.set_title('Reactor Power [MW] Trace', size=12)
23      pylab.setp(self.axes1.get_xticklabels(), fontsize=8)
24      pylab.setp(self.axes1.get_yticklabels(), fontsize=8)
25
26  def create_plot_panel(self):
27      """Embed matplotlib canvas in wxPython panel"""
28      self.init_plot()
29      self.canvas = FigCanvas(self.m_panel2, -1, self.fig)
```

Listing 22: Matplotlib Plot Setup

6.4 Real-Time Plotting

```python
 1  def draw_plot(self):
 2      """
 3      Update plots with latest reactor data
 4      Implements zoom functionality and dual-axis temperature plotting
 5      """
 6      # Determine plot window size based on zoom level
 7      zoomPercentage = self.zoom / 100.
 8      if zoomPercentage < 0.02:
 9          zoomPercentage = 0.02  # Minimum 2% zoom (2 seconds)
10
11      # Calculate number of data points to display
12      plotMask = int(zoomPercentage * len(self.data[0]))
13
14      # Create time axis
15      xdata = np.array(np.array(range(plotMask)) / float(plotMask)) * \
16              self.legoReactor.maxTime * zoomPercentage
17
18      # Extract data traces
19      pwrdata = qFuel(self.data[1][0, :][-plotMask:]) / 1.e6  # Power [MW
     ]
20      fuelTdata = self.data[1][2, :][-plotMask:]  # Fuel temp [K]
21      coolTdata = self.data[1][3, :][-plotMask:]  # Coolant temp [K]
22
23      # Clear previous plots
24      self.axes1.clear()
```

```python
25    self.axes2.clear()
26    self.axes3.clear()
27
28    # Set axis limits
29    self.axes1.set_ylim(0, 650.)   # Power: 0-650 MW
30    self.axes2.set_ylim(400, 1700.)   # Fuel temp: 400-1700 K
31    self.axes3.set_ylim(400, 700.)   # Coolant temp: 400-700 K
32
33    # Plot power
34    self.axes1.set_title('Reactor Power [MW] Trace', size=12)
35    self.axes1.set_ylabel('Power [MW]')
36    self.axes1.set_xlabel(str(round(max(xdata), 0)) + ' time [s]')
37    self.axes1.plot(xdata, pwrdata, linewidth=2, color='blue')
38
39    # Plot temperatures on dual y-axes
40    self.axes2.set_ylabel('Fuel Temperature [K]')
41    self.axes3.set_ylabel('Coolant Temperature [K]')
42    self.axes3.yaxis.set_label_position('right')
43    self.axes3.yaxis.tick_right()
44
45    # Red line for fuel temperature (left axis)
46    fuelPlot, = self.axes2.plot(xdata, fuelTdata, color='r',
47                                linewidth=2, label='Fuel T')
48    # Blue line for coolant temperature (right axis)
49    coolPlot, = self.axes3.plot(xdata, coolTdata, color='b',
50                                linewidth=2, label='Coolant T')
51
52    # Add legends
53    handles, labels = self.axes2.get_legend_handles_labels()
54    self.axes2.legend(handles, labels, loc=2)   # Upper left
55    handles, labels = self.axes3.get_legend_handles_labels()
56    self.axes3.legend(handles, labels, bbox_to_anchor=(0.402, 0.85))
57
58    # Render updated plot
59    self.canvas.draw()
```

Listing 23: 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

```python
1  def pauseSim(self, event):
2      """Pause/unpause simulation"""
3      self.paused = not self.paused
4
5  def SCRAM(self, event):
6      """SCRAM button: toggle reactor shutdown"""
7      self.scramToggle = not self.scramToggle
8      self.legoReactor.SCRAM(bool(self.scramToggle))
9
10 def pwrCtrlON(self, event):
11     """Toggle automatic power control mode"""
12     pwrSet = self.pwrSetPt.GetValue()
```

```python
13      self.pwrCtrlToggle = not self.pwrCtrlToggle
14      self.legoReactor.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.legoReactor.togglePwrCtrl(float(pwrSet))
21
22  def setRodPos(self, event):
23      """Set rod position from text entry"""
24      enteredVal = self.rodSetPt.GetValue()
25      self.legoReactor.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.legoReactor.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      self.legoReactor.setCoolantRate(float(self.coolantBox.GetValue()) *
    1.e3)
```

Listing 24: GUI Event Handler Implementations

6.6   Monitor Updates

```python
1  def updateMonitors(self):
2      """
3      Update all text displays and gauges with current reactor parameters
4      Called every redraw cycle (1 Hz)
5      """
6      # Rod position display
7      self.rodPosOut.SetValue(str(round(self.legoReactor.S[9], 1)))
8
9      # Temperature displays
10     self.cooltOut.SetValue(str(round(self.legoReactor.S[8], 1)))
11     self.fueltOut.SetValue(str(round(self.legoReactor.S[7], 1)))
12
13     # Power display
14     self.powOut.SetValue(str(round(float(qFuel(self.legoReactor.S[0]) /
    1.e6), 1)))
15
16     # Visual rod position gauge
17     self.rodGauge.SetValue(int(self.legoReactor.S[9]))
18
19     # Calculate and display coolant flow rate based on power
20     currentPower = float(qFuel(self.legoReactor.S[0]) / 1.e6)
```

```
21      maxPwr = 500.  # Maximum expected power [MW]
22      normPwr = abs(currentPower / maxPwr)
23      normPwr = 250. * normPwr
24      if normPwr >= 250:
25          normPwr = 250
26
27      # Map power to coolant flow rate (mimics Arduino motor control)
28      # Higher power      higher required flow rate
29      minFlowRate = 200.0  # kg/s at minimum power
30      maxFlowRate = 1800.0  # kg/s at maximum power
31      coolantFlowRate = minFlowRate + (maxFlowRate - minFlowRate) * (
    normPwr / 250.0)
32      self.coolantBox.SetValue(str(round(coolantFlowRate, 2)))
```

Listing 25: Update Display Monitors with Current State

6.7   Arduino Serial Communication

```
1  def writeToArduino(self):
2      """
3      Send reactor state to Arduino for physical model control
4      Updates servo (rod position), LED (power), and SCRAM indicator
5      """
6      if self.ser:
7          # ===== CONTROL ROD POSITION =====
8          # Map rod position (0-100%) to servo angle (5-140 degrees)
9          rodWriteOut = abs((self.legoReactor.S[9] / 50.) * 160.)
10         if rodWriteOut < 5.0:
11             rodWriteOut = 5.0  # Minimum servo position
12         elif rodWriteOut > 140.0:
13             rodWriteOut = 140.  # Maximum servo position
14
15         # Send command: 'r' + angle
16         self.ser.write(("r" + str(int(rodWriteOut))).encode())
17         time.sleep(0.1)  # Arduino processing time
18
19         # ===== REACTOR POWER LED =====
20         # Map power to LED brightness (0-255)
21         maxPwr = 500.  # Maximum power for full brightness [MW]
22         normPwr = abs(qFuel(self.legoReactor.S[0]) / 1.e6 / maxPwr)
23         normPwr = 250. * normPwr
24         if normPwr >= 250:
25             normPwr = 250
26
27         # Send command: 'p' + brightness
28         self.ser.write(("p" + str(int(normPwr))).encode())
29         time.sleep(0.1)
30
31         # ===== SCRAM INDICATOR LED =====
32         # Red LED: on during SCRAM, off otherwise
33         scramValue = 1 if self.legoReactor.scramToggle else 0
34
35         # Send command: 's' + state
36         self.ser.write(("s" + str(int(scramValue))).encode())
37         time.sleep(0.1)
```

Listing 26: Arduino Hardware Feedback Interface

Communication Protocol: The simple character-based protocol ('r', 'p', 's' commands) provides robust serial communication. Short delays (0.1s) ensure the Arduino has time to process commands and move actuators before the next command arrives.

6.8   Serial Port Auto-Detection

```python
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:
```

```
47          print("Arduino Not Detected. Running without serial connection"
        )
48      else:
49          print("Connection to Arduino Established on " + ser.port)
50
51      return ser
```

Listing 27: 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.

6.9   Application Entry Point

```
1 def main():
2     """
3     Application entry point
4     Creates wxPython application and main window
5     """
6     app = wx.App(False)  # Don't redirect stdout/stderr to GUI
7     frame = CalcFrame(None)
8     frame.Show(True)
9     app.MainLoop()  # Start event loop
10
11 if __name__ == "__main__":
12     main()
```

Listing 28: 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

```
1 #include <Servo.h>
2
3 Servo myServo;
4 int servoPin = 9;          // Control rod servo motor
5 int ledRGBRedPin = 11;     // SCRAM indicator LED (red)
6 int ledRGBBluePin = 6;     // Power indicator LED (blue)
7 int motorPWM = 3;          // Coolant pump motor PWM control
8
9 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 29: 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 2: 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

```
1  void loop(void){
2    int ledPowerSetting;
3    int rodHeightSetting;
4    int scramCondition;
5    int motorSpeed;
6
7    if (Serial.available() > 0)
8    {
9      char inByte = Serial.read();
10
11     switch(inByte)
12     {
13     case 'p': // Power command
14       // Set blue LED brightness (0-255)
15       ledPowerSetting = numberFromSerial();
16       analogWrite(ledRGBBluePin, ledPowerSetting);
17
18       // Map power (0-255) to motor speed (20-180)
19       // Minimum speed 20 keeps motor running smoothly
20       // Maximum speed 180 prevents overdriving
21       motorSpeed = map(ledPowerSetting, 0, 255, 20, 180);
22       analogWrite(motorPWM, motorSpeed);
23       break;
24
25     case 'r': // Rod position command
26       rodHeightSetting = numberFromSerial();
27       myServo.write(rodHeightSetting);  // Move servo to angle
28       break;
29
30     case 's': // SCRAM condition
31       scramCondition = numberFromSerial();
32       Serial.print("SCRAM received: ");
33       Serial.println(scramCondition);
34
35       if (scramCondition > 0) {
36         // SCRAM active: red LED on at full brightness
37         analogWrite(ledRGBRedPin, 255);
38         scramActive = true;
39       } else {
40         // SCRAM cleared: red LED off
41         analogWrite(ledRGBRedPin, 0);
42         scramActive = false;
43       }
44       break;
45     }
46
47     Serial.flush();  // Clear serial buffer
48   }
49  }
50
51  int numberFromSerial(void)
52  {
53    /*
54     * Parse integer from serial buffer
55     * Reads digits until end of number or buffer full
56     */
```

```
57    char numberString[8];
58    unsigned char index=0;
59    delay(10);  // Wait for full number to arrive
60
61    while(Serial.available() > 0)
62    {
63       delay(10);
64       numberString[index++]=Serial.read();
65       if(index>6)
66       {
67          break;  // Limit to 6 digits
68       }
69    }
70    numberString[index]=0;  // Null terminator
71    return atoi(numberString);  // Convert string to integer
72 }
```

Listing 30: Arduino Main Loop and Command Parser

Command Protocol Summary:

Table 3: 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:

$$\frac{dn}{dt} = \frac{\rho(t) - \beta}{\Lambda} n + \sum_{i=1}^{6} \lambda_i C_i \tag{14}$$

$$\frac{dC_i}{dt} = \frac{\beta_i}{\Lambda} n \cdot \eta - \lambda_i C_i \quad (i = 1, \ldots, 6) \tag{15}$$

$$\frac{dT_f}{dt} = \frac{Q(n) - hA_c(T_f - T_c)}{m_f C_{p,f}(T_f)} \tag{16}$$

$$\frac{dT_c}{dt} = \frac{hA_c(T_f - T_c) + C_{p,c}(T_{in} - T_c)\dot{m}_c}{m_c C_{p,c}} \tag{17}$$

$$\frac{dh}{dt} = \dot{h}(t) \tag{18}$$

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

8.2 Reactivity Functions

8.2.1 Control Rod Worth

Differential rod worth (sinusoidal):

$$R(h) = k \sin\left(\frac{\pi h}{100}\right) \tag{19}$$

Integral rod worth:

$$\rho_{rod}(h) = \int_0^h R(h') \, dh' = -\frac{100k}{\pi} \left[\cos\left(\frac{\pi h}{100}\right) - 1\right] \tag{20}$$

8.2.2 Temperature Reactivity

$$\rho_{temp}(T_f) = \alpha_T(T_f - T_{in}) \tag{21}$$

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:

$$\frac{d(\delta n)}{dt} \approx \frac{\rho_0 - \beta}{\Lambda}\delta n + \frac{n_0}{\Lambda}\delta\rho + \sum_{i=1}^{6}\lambda_i\delta C_i \tag{22}$$

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 $\to \rho \uparrow$

2. Neutron population increases $\to n \uparrow$

3. Power increases $\to Q = Q(n) \uparrow$

4. Fuel temperature rises $\to T_f \uparrow$

5. Negative temperature feedback $\to \rho \downarrow$

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:

$$\tau_{feedback} \approx \frac{m_f C_{p,f}}{hA_c + |n_0 \alpha_T Q'/\rho|} \tag{23}$$

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 legoReactor.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).

9.3   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

- 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. **No Xenon Poisoning**: Omits Xe-135 buildup/decay (important for long-term operation)

   - Valid for short transients (< 1 hour)
   - Xenon effects critical for load-following operations
   - Samarium-149 poisoning also neglected

5. **No Burnup**: Fuel composition doesn't change (valid for short demonstrations)

   - No plutonium buildup or U-235 depletion
   - Reactivity coefficients remain constant
   - Appropriate for training scenarios of fixed duration

6. **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. **Xenon Dynamics**: Add Xe-135 and Sm-149 poisoning effects

   - Demonstrate load-following challenges
   - Show xenon oscillations in spatial kinetics

2. **Multi-Region Model**: Implement core, reflector, and pressure vessel regions

   - More accurate neutron leakage
   - Region-dependent temperature effects

3. **Improved Thermal-Hydraulics**:

   - Axial temperature profiles
   - Two-phase flow modeling (boiling)

- DNB (Departure from Nucleate Boiling) limits

4. **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:

$$\frac{dn(t)}{dt} = \frac{\rho(t) - \beta}{\Lambda} n(t) + \sum_{i=1}^{I} \lambda_i C_i(t) \tag{24}$$

$$\frac{dC_i(t)}{dt} = \frac{\beta_i}{\Lambda} n(t) - \lambda_i C_i(t) \quad (i = 1, \ldots, I) \tag{25}$$

A.2   Reactor Period

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

$$T = \frac{\Lambda}{\rho - \beta} \quad \text{(for } \rho < \beta) \tag{26}$$

A.3   Prompt Jump Approximation

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

$$\frac{n(t = 0^+)}{n(t = 0^-)} \approx \frac{\rho_0 + \Delta\rho}{\rho_0} \tag{27}$$

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: $k = \frac{1}{7.5408 \times 10^{-3} + 1.7692 \times 10^{-5} T + 3.6142 \times 10^{-9} T^2}$ 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/