Simulation of Time-Dependent Neutron Populations for Reactor Physics Applications using the Geant4 Monte Carlo Toolkit

SIMULATION OF TIME-DEPENDENT NEUTRON POPULATIONS FOR REACTOR PHYSICS APPLICATIONS USING THE GEANT4 MONTE CARLO TOOLKIT

BY

LIAM RUSSELL, B.A.Sc., (Engineering Physics)

University of British Columbia, Vancouver, Canada

A thesis

submitted to the department of Engineering Physics

and the School of Graduate Studies

of McMaster University

in partial fulfilment of the requirements

for the degree of

Master's of Applied Science

McMaster University

Master's of Applied Science (2012) McMaster University

(Engineering Physics) Hamilton, Ontario, Canada

TITLE:	Simulation of Time-Dependent Neutron Populations for Reactor Physics Applications using the Geant4 Monte Carlo Toolkit
AUTHOR:	Liam Russell B.A.Sc., (Engineering Physics) University of British Columbia, Vancouver, Canada
SUPERVISOR:	Dr. A. Buijs
NUMBER OF PAGES:	xv, 194

Abstract

When the material or geometry of a reactor varies with time, the neutron flux will respond in the form of a reactor transient. These transients can occur during normal operations when control rods are moved or the reactor is refuelled (CANDU). During a reactor accident, the transient response is especially important because the reactor properties vary quickly with large amplitudes. Therefore, better understanding these conditions allows for improved identification, prevention and mitigation of reactor transients. However, current nuclear simulation codes are generally limited in their ability to model transient behaviour.

The NStable code was created to model time-dependent neutron populations in multiplying mediums using the Geant4 Monte Carlo toolkit. The neutron population is allowed to evolve in time, but is periodically renormalized so that the total number of neutrons is constrained within a manageable range. This ensures that the simulation is viable even in highly sub- or supercritical environments. Since Geant4 was not intrinsically designed to track a neutron population over "long" time periods (up to 10 s), the population renormalization mechanisms needed to be created and integrated with Geant4. Additionally, nuclear reactor analysis functionality was added to calculate important quantities such as k_{eff}.

The NStable code was validated using three established nuclear simulation codes: MCNP 5, DRAGON 3.06J, and TART 2005. The validation cases compared spatial distributions and criticality estimates for either homogeneous spheres (uranium-235 or a uranium-heavy water mixture) or the standard CANDU 6 lattice cell. For all three

systems, the criticality estimates in NStable agreed with the appropriate validation code within 10 mk (TART for the spheres and DRAGON for the CANDU 6 lattice). Finally, the NStable code was also used to simulate a temperature transient in a UHW sphere where the temperature linear increased by 700 K over 50 ms. In response to the increasing temperature, k_{eff} decreased by 100 mk over the same period. In the future, transient modelling in NStable should be investigated further to reproduce actual experimental results, and to couple NStable with a thermohydraulics code to simulate a full transient response.

Acknowledgements

First, I would like to thank my co-supervisors, Dr. Adriaan Buijs and Dr. Guy Jonkmans, for their help and support over the past two years. I would also like to thank Bruce Wilkins at AECL Chalk River for offering his expertise in the area of Monte Carlo simulations, and Dr. John Luxat for allowing me to use his computation cluster at McMaster, which was the workhorse behind the results in this thesis.

I would also like to thank everyone in nuclear engineering at McMaster. You guys provided a great deal of help and advice, as well as welcome distractions when the frustrations of coding, debugging, and writing got to be too much. In particular, I want to acknowledge the resident Ph.D.'s - David Hummel, Ken Leung, Andrew Morreale and (Dr.) Matt Ball - who were always willing to talk through any obstacles I encountered in my research. Additionally, I would like to offer a special thanks to Wesley Ford who worked for Dr. Buijs and myself as a summer student in the last four months of my thesis. His help was invaluable and ensured that I finished this thesis on time.

Finally, I would like to thank my brother, who, in addition to everything else, was my personal go to resource for anything coding related. Moreover, I could always count on being able to crash at his place when I needed a break for a couple of days.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since the "nuclear age" started in the late 1950's, computer simulations have become increasingly important to the design, utilization and safety of nuclear devices. Given the extreme cost of nuclear facilities, especially power reactors, computer simulations provide a relatively cheap and effective alternative to experimental prototypes. Although the need for experimental facilities is unlikely to ever be eliminated, computer simulations have become one of the main tools for nuclear researchers and the nuclear industry. Coupled with the rapid development of computers, advances in computational nuclear simulations are allowing researchers to model difficult problems, such as accident scenarios, where many coupled systems change rapidly over a short duration.

Due to increased awareness of the effects of radiation, especially on biological systems, as well as increased knowledge of nuclear power accident scenarios from the accidents at Three Mile Island, Chernobyl and Fukushima, safety is a priority for the nuclear community. Given the complexity of nuclear reactors and the speed with which accident conditions evolve, effectively modelling these conditions is difficult. However, any improvements in the understanding of such conditions, especially how

they evolve, could produce many benefits. Not only would a better understanding improve safety overall, but it would also improve the ability to recognize, prevent and mitigate accidents. Furthermore, an improved understanding could allow the current safety cases that guide reactor regulations to be refined to the benefit of both the public, in terms of increased safety, and the nuclear industry, in terms of reduced costs.

Modelling dynamic reactor conditions, known as reactor kinetics, also has more routine applications. Most reactors are modelled as essentially static with respect to short time scales (less than an hour). However, the properties and outputs of a reactor are constantly fluctuating within the operating limits. This nuclear noise can come from many sources, such as the coolant flow, and must be actively controlled by the reactor regulating systems [1]. Additionally, if the differences in fuel composition across the core are not properly managed, they can lead to power oscillations between two regions of the reactor (e.g. xenon oscillations). In all of these cases, a better understanding of the reactor kinetics could improve the response and effectiveness of the reactor control systems.

Traditionally, nuclear simulation codes (programs) have focused on static cases, such as the power levels and neutron flux of a reactor at full power. In these cases, the system is mostly stable in time, and can be assumed to be time-independent without a major loss of accuracy. Such calculations have been used in the nuclear industry to not only design reactors, but also to maintain reactor performance in areas such as fuelling schemes (how and when to fuel a reactor). Additionally, deterministic time-dependent reactor calculations can be achieved using either the point kinetics approximation or a space-energy dependent dynamics approach [2]. The simpler

point kinetics approximation, or point reactor model, assumes the neutron spatial distribution is constant in time. This assumption is insufficient for many accident scenarios, and thus, more complicated space-energy dependent dynamics approaches are employed instead. These approaches, which include finite difference methods and modal approaches, are more accurate but are also more time consuming [2]

Accident conditions are transient, and therefore, must be modelled as dynamic systems. Monte Carlo simulations, which follow neutrons (and other particles) in space, time and energy, provide an excellent platform for such dynamic simulations. At the most basic level, Monte Carlo simulations are simple; they faithfully reproduce the given nuclear data through random sampling. As such, these simulations can be applied to complex systems without many of the approximations necessary for other simulation methods, such as spatial and energy disretization. However, this fundamental simplicity also causes Monte Carlo simulations to be computationally expensive in terms of time and resources because every neutron and secondary particle is followed from birth to death or escape. Moreover, the random sampling is susceptible to statistical errors, which may only be eliminated if an infinite number of neutrons are followed over the course of the simulation. Fortunately, the improvements in computer processing power and simulation algorithms have relieved some of these burdens so that Monte Carlo simulations are a viable alternative in dynamic simulations.

1.1 Objectives

The objectives of the project detailed in this thesis were

1. To model time-dependent neutron populations with the Geant4 Monte Carlo

toolkit.

2. To allow the material and geometric properties of the model to change with respect to time.

3. To use the evolution of the neutron population to calculate macroscopic characteristics [1] (e.g. k_{eff}).

4. To use the evolution of both the neutron population and model properties (material and geometric) to simulate reactor transients.

The Geant4 (GEometry ANd Tracking 4) Monte Carlo toolkit (explained in Section 1.2) served as the base for the simulations in this project [3]. However, Geant4 was not designed specifically for use in reactor simulations, and therefore, most of the project focused on developing and validating the computer code necessary to extend Geant4 for this purpose.

1.2 Geant4

Geant4 is an open-source physics simulation toolkit that was developed by the Geant4 Consortium. This international consortium is comprised of scientists from many research institutions, including the European Organization for Nuclear Research (CERN), the Stanford Linear Accelerator (SLAC), TRIUMF (Cananda), KEK (Japan), the University of Alberta, and the Massachusetts Institute of Technology (MIT) [3]. The toolkit was designed to be applicable over a diverse range of problems, from small

[1] *Macroscopic characteristics* refers to characteristics that apply to the simulation as a whole. These are also referred to as integral characteristics.

fundamental physics simulations to full-scale detector simulations for particle accelerators such as the LHC (Large Hadron Collider). Moreover, Geant4 embraces customization, and encourages users to develop algorithms and functions for their own applications. It also is able to simulate a wide range of particles, such as neutrons, photons and neutrinos, and physical interactions, such as hadronic and electromagnetic interactions [3].

1.2.1 History

In 1993, two separate studies examined how modern computing techniques could be applied to particle physics simulations to improve upon GEANT3 (the previous version of GEANT). These studies by KEK and CERN merged and concluded that a new version of GEANT should be developed using object-oriented C++. This led to the creation of the RD44 project, spearheaded by CERN, which was mandated to outline and design Geant4. In 1998, the first production release of Geant4 was completed and made available. Subsequently in 1999, the development and maintenace of Geant4 was passed to the newly formed Geant4 Consortium [3].

The consortium is organized into a Collaboration Board, a Technical Steering Board, and several working groups, each of which is responsible for an area of physical phenomena, such as hadronic interactions. The Collaboration Board is mandated to manage the resources and agreements of the consortium, while the Technical Steering Board decides on the manner in which physics phenomena are implemented, and prioritizes future development. Additionally, the Technical Steering Board appoints coordinators to supervise each working group. The working groups themselves are responsible for the development and maintenance of the code library covering their

chosen physical phenomena [3].

1.2.2 Design Philosophy

Unlike most nuclear simulation codes, Geant4 was not designed to be a complete program for the user to interact with. Instead, Geant4 is a "toolkit" of components that the user can use to build a simulation. To this end, Geant4 was designed to be modular and flexible, so that users could assemble a program at compile-time out of components that either come from the toolkit or are self-supplied. The toolkit components may even be supplanted by user-supplied code, making Geant4 extensible [3].

Openness was also an important consideration during the design process of Geant4. In particular, care was taken to make the implementation of the physics models open and transparent. Additionally, end-users are given a choice of physics models, and the opportunity to use custom models that they have designed or modified themselves. The nuclear data is separate from the physics models so that the user may chose their own nuclear data library. Finally, not only is the source code freely available from the Geant4 website, but also several online code browsers were created to help users to navigate the source code [3]. This is extremely beneficial when attempting to follow algorithms or processes in the source code.

1.2.3 Structure

The Geant4 toolkit is composed of three major parts: the source code, the nuclear data, and the utility files, which aid the user in creating working programs (e.g. makefiles). Upon installation, the Geant4 source code is compiled into libraries that are linked to the user's code at compile-time to form fully functional programs. In

general, every major component of Geant4 is a C++ class, and these classes often contain member data objects created from other classes in the Geant4 source code[2].

1.2.3.1 Organization

Figure 1.1 shows the general hierarchy and progression of data in Geant4. The lines denoted data flow toward the end of the link with the circle (e.g. *Tracking* uses data from *Hits* and *Processes*, and then passes the tracking data on to *Event*). The data progression was designed to be one way to prevent any chance of circular dependencies in the core Geant4 classes (e.g. a class A has an object of class B, but class B calls a member function of class A, which will confuse the compiler). The categories of classes in the figure represent the major components of a Geant4 simulation, and the ones most relevant to this thesis are defined below

Hit An interaction involving a tracked particle that invokes a physics process.

Process A physics model used to calculate the likelihood and result of a hit. Models include hadronic interactions (scattering, fission, radiative capture), transportation, and user defined processes.

Step A single discrete movement of a particle, which starts and ends with a hit.

Track A series of sequential steps that make up the history of the particle.

Primary An initial source particle used to start an event.

Event The entire history of n primaries and their descendants from birth until death by absorption or escape from the simulation geometry.

[2]Geant4 is written in C++, so the discussion of the Geant4 source code and the extensions made in this project will make use of general C++ nomenclature and concepts. For a short discussion on the basic C++ concepts that are used heavily in this project, see Appendix A.

Run A collection of independent events.

Simulation The entire modeling process from start to finish, which may include multiple runs.

The most fundamental structure in every Geant4 simulation is the *kernel*. It is responsible for tracking the particles with respect to the physics processes and simulation geometry. It achieves this using *manager* classes to handle each step in the tracking process [3]. These managers include

- Run manager
- Event manager
- Tracking manager
- Stepping manager
- Process manager

These are singleton classes (only one instance allowed) that can be called at any point in the simulation. They allow access to the class category they manage - the process manager allows access to the processes defined in the simulation - and they carry out any actions necessary for tracking a particle. For example, the stepping manager simulates each step of a particle, and does so with track and process information from the tracking and process managers respectively.

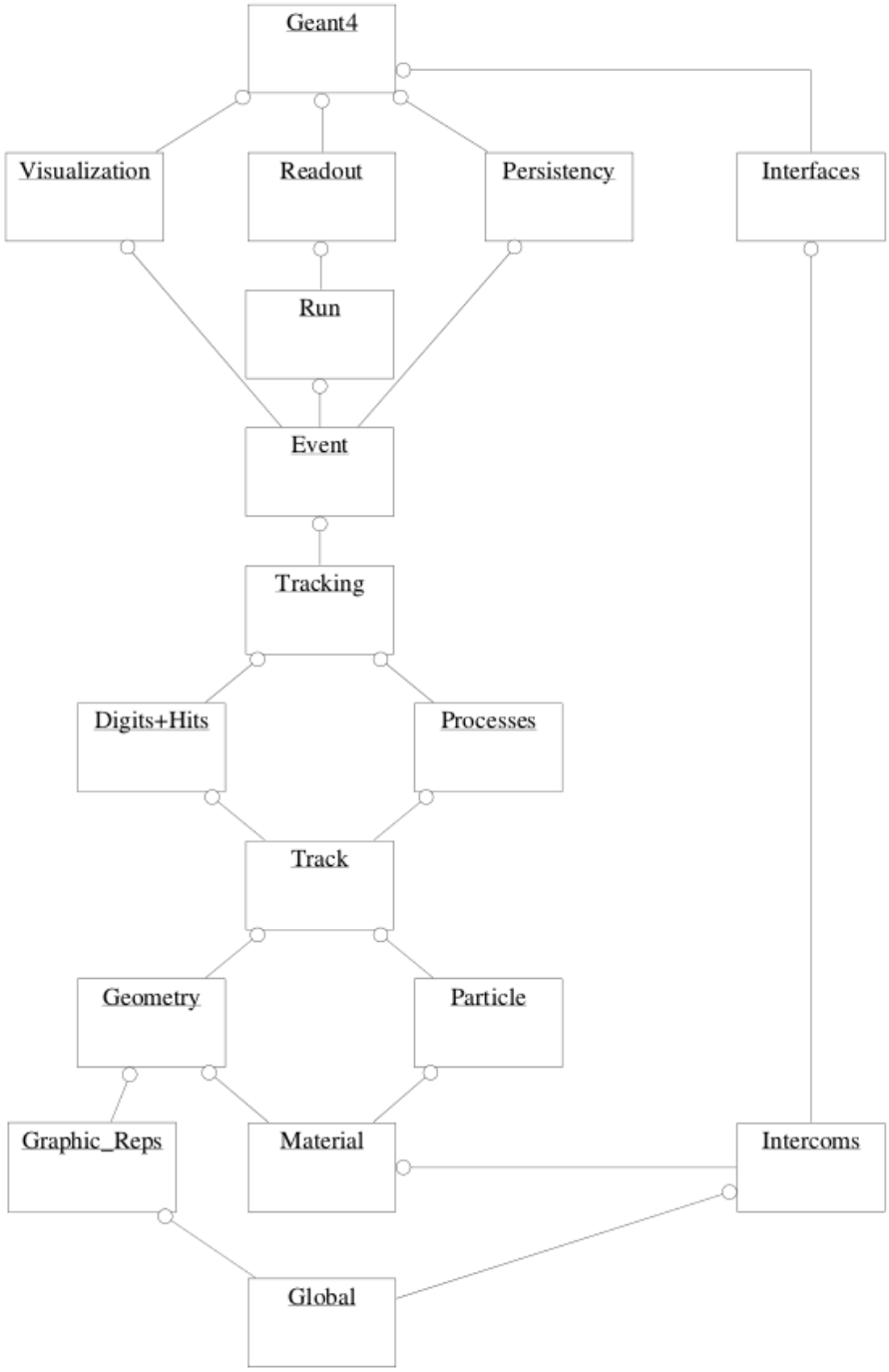


Figure 1.1: Geant4 simulation hierarchy [3].

1.2.4 Creating a Geant4 Simulation

To create a Geant4 simulation, the user is required to provide the following classes

World constructor

Defines the material and geometric composition of the simulation world [3].

Primary generator

Defines source distribution of primary particles.

Physics list

Defines the physics processes to be used in the simulation.

The user may also provide the following optional action classes

Run action

Defines the actions to be taken at the beginning and end of each run.

Event action

Defines the actions to be taken at the beginning and end of each event.

Tracking action

Defines the actions to be taken at the beginning and end of tracking for the current track.

Stepping action

Defines the actions to be taken at the beginning and end of each step.

The user must also provide a driver file containing the *main* function. The driver file will instantiate the kernel, the run manager and the user defined classes. Once all the

[3] The *simulation world* is the "physical" environment that the neutrons move through during a simulation. It includes *all* of the geometric structures, and associated materials, that have been defined for the simulation.

necessary classes and functions have been defined, the user-generated source files are compiled and linked to the Geant4 source code libraries to form a working program.

While the above classes must be user-generated, this does not mean that every detail must be explicitly coded by the user. The Geant4 toolkit provides many helpful classes that can be used to quickly build user-generated classes. For example, the primary generator class can be quickly created using the *G4ParticleGun* class. The user simply needs to create a particle gun object and specify the number, type, position and momentum of the primary particles.

1.2.5 Suitability of Geant4

Geant4 was chosen for this project due to its ability to be adapted to most problems. While other, more established codes are appropriate for their intended purpose, such as MCNP (Monte Carlo N-Particle) for multiplication eigenvalue calculations (see Section 2.2.2), these codes are restrictive in terms of access to simulation data [4]. In general, the developers built these other codes for a given set of tasks with a limited set of output options, and therefore, applying these codes to problems outside of their scope is challenging. Conversely, Geant4 makes few assumptions about the end-user's intended application, and provides a framework to build new simulation methods. The benefit of this flexibility is tempered by the increased effort needed to build a complete simulation, but in the case of this project, other codes, such as MCNP, would have resulted in inelegant solutions created by "hacking" the functionality of the established code.

While Geant4 is not an established code in the field of reactor research, it is in many other fields. Geant4 is extensively used in high energy particle physics, and

was used to design the detectors for the LHC. Additionally, it has seen constant development and improvement over its 14 year lifespan, with a large community of active users who may be approached in the case of technical problems [3]. Therefore, while the nuclear engineering community may be unfamiliar with Geant4, it has been extensively validated in many other areas [5, 6, 7, 8].

1.3 Summary

This thesis covers the development of algorithms and procedures for the Geant4 Monte Carlo toolkit to simulate time-dependent neutron populations in environments where the material or geometric composition may also change with time. Chapter 2 covers the relevant background information necessary to inform the reader for the discussion in the subsequent chapters. Chapter 3 examines Monte Carlo nuclear simulation codes that are similar to the code developed in this project, as well as some other applications of Geant4 in reactor research. The specific contributions and code development for this project are described in Chapter 4, and Chapter 5 covers the validation of the Geant4 reactor simulation method against standard reactor simulation codes. Chapters 6 contains conclusions that were reached at the end of this project and possible future extensions. Lastly, the conclusions are followed by the references and the appendices.

Chapter 2

Background and Theory

The scope of this chapter extends from the fundamental neutron transport equation to solutions of the transport equation by deterministic and stochastic methods. In particular, this section will focus on stochastic methods of solution, namely Monte Carlo methods, including general Monte Carlo solution methods and specific implementations used in Geant4. The Geant4 discussion is limited to the base libraries provided by the Geant4 consortium; see Chapter 4 for the contributions to the Geant4 code base made in this project.

2.1 Neutron Transport

2.1.1 Neutron Interactions

Neutron interactions are classified into two major categories, absorption and scattering [1]. Scattering interactions occur when a neutron "collides" (interacts) with a nucleus but is not (permanently) absorbed. Instead, the momentum of the neutron

changes depending on the nature of the collision. In many scattering collisions, the neutron is temporarily absorbed by the nucleus to form an excited compound nucleus, which decays and reemits a neutron. These interactions can either be elastic or inelastic (see below), and normally occur for neutrons with energies above 10 keV because the incident neutron must have sufficient energy to raise the nucleus to the first excited state. Neutrons may also scatter elastically off the nuclear potential of the nucleus; in this case, the neutron never penetrates the nuclear surface [1].

Elastic collisions (n,n) Kinetic energy is conserved in elastic collisions. Energy is transferred between the neutron and the nucleus, and the direction of the neutron is changed but no additional particles are created.

Inelastic collisions (n,n') Inelastic collisions do not conserve kinetic energy and some of the kinetic energy of the collision is released as radiation.

The two major absorption interactions are radiative capture and fission. In both cases, the neutron is absorbed by the nucleus, creating an unstable compound nucleus [1].

Radiative capture (n,γ) The excited compound nucleus decays by emitting gamma rays, but the incident neutron remains.

Fission (n,f) The compound nucleus splits into two smaller nuclei and two to three fission neutrons. The daughter nuclei (fission products) are generally unstable and decay over time by particle emission (e.g. beta, gamma, neutron, neutrino).

Only fissionable isotopes undergo fission, which includes all isotopes with an atomic number greater than 89. Furthermore, fission only occurs if the captured neutron provides enough energy to overcome the binding energy of the nucleons in the nucleus

(this energy is known as the fission threshold and is approximately 6-9 MeV). The incident neutron contributes its kinetic energy, as well as the binding energy gained through its capture. For fissile isotopes, such as uranium-235, this binding energy alone is sufficient to cause fission, but for isotopes that are only fissionable, such as uranium-238, additional energy is needed to reach the fission threshold. For uranium-238 to fission due to neutron capture, this energy must be provided by the kinetic energy of the incident neutron. Since nuclear fission only requires sufficient energy to surpass the fission threshold, fission may also be induced by high energy photons [1].

Other absorption interactions can release particles such as alpha particles and neutrons. The neutron releasing interactions (not including fission) are called (n,*n) interactions where '*' is the number of neutrons at the end of the interaction. Additionally, photons may also produce *photo-neutrons* through (γ,n) interactions. These neutron-producing interactions are particularly important to reactor physics because they contribute positively to the overall neutron economy.

2.1.2 Interaction Cross Sections

The propensity of a neutron to interact with a given material depends on the microscopic cross sections for that neutron-material pair. The microscopic cross section is defined as the "effective cross-sectional area per nucleus seen by the [neutron]" and is usually given in units of barns ($10^{-24}\,cm^2$) [9]. This is not the actual cross sectional area of the nucleus, but rather, a physical quantity that depends on the atomic number and mass of the nucleus, and the energy of the incoming neutron. The probability of an interaction between the neutron and the material depends also on the density of the material. This probability per unit length is defined as the macroscopic cross

section and is given by [1]

$$\Sigma_i(E) = N\sigma_i(E) = \frac{N_A\rho}{M}\,\sigma_i(E) \tag{2.1}$$

where N is the atomic number density of the material, N_A is Avogadro's number, ρ is the density of the material, M is the molar mass of the material, and i denotes the type of interaction. Since the cross sections correspond to independent neutron interactions, the overall effect of multiple cross sections is a superposition of the individual cross sections. That is,

$$\begin{aligned} \sigma_t &= \sigma_s + \sigma_a = \sigma_\gamma + \sigma_f + \sigma_e + \sigma_{in} + ... \\ \Sigma_t &= \Sigma_s + \Sigma_a = \Sigma_\gamma + \Sigma_f + \Sigma_e + \Sigma_{in} + ... \end{aligned} \tag{2.2}$$

where the interaction types are as follows: t is total, s is scattering, a is absorption, e is elastic, and in is inelastic. The extension from micro- to macroscopic cross sections in Equation 2.2 relies upon Equation 2.1, where the atomic number density, N is common to all interaction types for a given isotope.

The fundamental neutron cross sections are defined for a given particle, interaction and isotope. For materials containing multiple isotopes, the total macroscopic cross section must be formulated using the interaction and neutron densities for each isotope present. That is

$$\Sigma_t = \sum_i \sum_j N_j \sigma_i^j \tag{2.3}$$

where j denotes the isotope and i denotes the neutron interaction type [1].

2.1.2.1 Mean Free Path

Another important quantity is the average distance a neutron will travel before interacting with a nucleus, which is known as the mean free path length, λ. This quantity can be defined for either an individual interaction or multiple interactions (see Equation 2.2). The mean free path length for an interaction i, is given by [1]

$$\lambda_i = \frac{1}{\Sigma_i} \tag{2.4}$$

which has units of length.

2.1.3 Neutron Transport Equation

The fundamental governing equation of reactor physics is the neutron transport equation [9]:

$$\begin{aligned}\left[\frac{1}{\upsilon}\frac{\partial}{\partial t} + \hat{\Omega}\cdot\overrightarrow{\nabla} + \Sigma_t(\overrightarrow{r},E)\right]\psi(\overrightarrow{r},\hat{\Omega},E,t) \\ = Q + \int dE' \int d\Omega'\, \Sigma_s(\overrightarrow{r},E'\rightarrow E,\hat{\Omega}'\cdot\hat{\Omega})\psi(\overrightarrow{r},\hat{\Omega}',E',t) \\ + \chi_p(E)\sum_i \left(1-\beta^i\right)\int dE' \int d\Omega'\, \nu\Sigma_f^i(\overrightarrow{r},E')\psi(\overrightarrow{r},\Omega',E',t) \\ + \sum_l \chi_l(E)\lambda_l C_l(\overrightarrow{r},t)\end{aligned} \tag{2.5}$$

The variables in Equation 2.5 are as follows:

- ψ is the neutron flux
- E is the neutron kinetic energy
- υ is the neutron speed $\left(\upsilon = \sqrt{2E/m_n}\right)$

- m_n is the neutron rest mass
- Ω is the neutron momentum direction (solid angle)
- Σ_t is the total interaction cross section
- Σ_s is the collision (scattering) cross section
- Σ_f is the fission cross section
- ν is the average neutron yield (multiplicity) per fission
- χ_p is the prompt fission neutron energy spectrum
- β^i is the delayed neutron fraction for delayed neutron precursor group i
- χ_l is the delayed neutron energy spectrum for group l
- λ_l is the group l decay constant
- C_l is the group l precursor concentration
- Q is any other neutron sources ($Q(\overrightarrow{r}, \hat{\Omega}, E, t)$)

Equation 2.5 is organized with neutron sinks on the left and neutron sources on the right, and the net neutron gain equals the time dependent rate of change of the neutron flux $\left(\frac{1}{v}\frac{\partial \psi}{\partial t}\right)$. Note that the neutron sources and sinks are defined for a particular position, energy, time and direction of flight of the neutron. For simplicity, the delayed neutron precursors, those fission products that decay and release neutrons at some time after the fission, are grouped according to the decay constant for each precursor. Thus, the precursor spectrum is reduced to l groups (normally six), whose concentration changes according to the Bateman equations [1]:

$$\frac{\partial}{\partial t}C_l(\overrightarrow{r}, t) = \sum_i \beta_l^i \int dE' \int d\Omega'\, \nu\Sigma_f^i(\overrightarrow{r}, E')\psi(\overrightarrow{r}, \Omega', E', t) - \lambda_l C_l(\overrightarrow{r}, t) \qquad (2.6)$$

where $\beta^i = \sum_l \beta_l^i$, λ_l is the average decay constant for group l, and i denotes a particular fissile isotope. Additionally, the external source, Q may also be configured such that the position, strength and direction of the source are time-dependent. For clarity, Equation 2.5 can be simplified using operator notation

$$\left(\frac{1}{\upsilon}\frac{\partial}{\partial t} + T - S - F\right)\psi \;=\; Q \;+\; \sum_l \chi_l(E)\lambda_l C_l(\overrightarrow{r}, t) \tag{2.7}$$

where the transport, removal (neutron sink) and fission operators are given below, respectively.

$$\begin{aligned}
T\psi &= \left(\hat{\Omega}\cdot\overrightarrow{\nabla} + \Sigma_t(\overrightarrow{r}, E)\right)\psi(\overrightarrow{r}, \hat{\Omega}, E, t) \\
S\psi &= \int dE' \int d\Omega'\, \Sigma_s(\overrightarrow{r}, E' \rightarrow E, \hat{\Omega}'\cdot\hat{\Omega})\psi(\overrightarrow{r}, \hat{\Omega}', E', t) \\
F\psi &= \chi_p(E)\sum_i \left(1-\beta^i\right)\int dE' \int d\Omega'\, \nu\Sigma_f^i(\overrightarrow{r}, E')\psi(\overrightarrow{r}, \Omega', E', t)
\end{aligned}$$

While Equation 2.5 is generally the most fundamental equation used in neutron transport, it does rely on the following assumptions [9]:

1. Particles may be considered as points.
2. Particles travel in straight lines between point collisions.
3. Particle-particle interactions may be neglected.
4. Collisions may be considered instantaneous.
5. Materials are assumed to be homogeneous and time-independent.

Assumptions 1, 3 and 4 rely on the small size of the neutron compared to atoms and interatomic spaces, the comparably low density of neutrons in interatomic spaces,

and the short time scale of nuclear reactions; with the exception of radioactive decay nuclear interactions like fission have a time scale of approximately 10 fs [1]. Assumption 2 is true for neutrons regardless of the presence of electromagnetic fields because neutrons have no electric charge, and other nuclear forces, such as the strong and weak nuclear forces, are negligible outside of a small region surrounding the nucleus, which is comparable to the atomic radius. Finally, assumption 5 is generally true for simulated reactor materials given the lack of detailed structure in materials in nuclear simulations, although some anisotropy with respect to the direction of travel of the neutron can be included [9]. In particular, the time scale of material property changes is on the order of milliseconds to days, whereas the time between neutron-nucleus interactions is on the order of nanoseconds to microseconds.

2.2 Reactor Kinetics

Reactor kinetics refers to the time-dependent changes in the neutron flux of the reactor. These changes can be described by Equations 2.5 and 2.6, although simplifications may be applied to reduce the complexity of the problem [9]. In the case of systems containing fissionable material, the fissioning of these materials will give rise to new generations of neutrons; these systems are said to be *multiplying mediums*. For reactors, the balance between neutron production through fission and neutron loss through absorption or escape is an extremely important state called criticality [9].

2.2.1 Criticality

The time-dependent evolution of a neutron population in a system is divided into three regimes. If the neutron population grows with time, it is said to be *supercritical*. If the neutron population is stable in time, then the system is *critical*, and finally, if the population shrinks over time, then the system is *subcritical*. The degree of criticality is measured either with the neutron multiplication factor, k_{eff}, or the reactivity, ρ, of the system. The neutron multiplication factor is defined as [1]:

$$k_{eff} \equiv \frac{\text{Rate of neutron production}}{\text{Rate of neutron loss}} = \frac{P(t)}{L(t)} \tag{2.8}$$

and the reactivity of the system is related to the criticality of the system by

$$\rho = 1 - \frac{1}{k_{eff}} \tag{2.9}$$

Thus, using the definitions for the three regimes of criticality defined above, we have

$$k_{eff}\ ,\ \rho = \begin{cases} > 1, & > 0 & \text{supercritical} \\ 1, & 0 & \text{critical} \\ < 1, & < 0 & \text{subcritical} \end{cases}$$

In a non-critical multiplying medium (without a significant external source), the neutron population will either increase or decrease exponentially depending on whether k_{eff} is greater or less than one [9]. This exponential population change occurs because equation Equation 2.5 has positive feedback through the neutron flux, ψ. The source and sink terms all depend positively on the flux such that any imbalance between the sources and sinks is amplified over time. Thus, the criticality of a system is especially

important to nuclear reactor operators; for a reactor to be stable, it must be critical or shutdown. When a reactor is starting up, shutting down or changing power levels, the reactor may briefly go into a sub- or supercritical state, but it must always return to a critical state when running [1]. The reactor is only left in a non-critical state in full shutdown (i.e. subcritical).

2.2.2 Time-Independent Eigenvalue Problems

Most numerical solutions of the neutron transport equation solve the time-independent form of Equation 2.5 [9].

$$(T - S - F)\,\psi \;=\; Q \tag{2.10}$$

Obviously, this equation is no longer true if the system is not critical because the time-derivative of the flux has been removed, so any difference between the strength of the sources and the sinks will no longer be balanced. Thus, Equation 2.10 must be modified to obtain a solution in these cases. These modifications, which take the form of eigenvalues, are useful to determine quantitatively how far the system is from critical. Two eigenvalue solutions, the multiplication eigenvalue and the time-absorption eigenvalue, will be discussed below. For simplicity, the external source term will also be neglected from the following discussion. Additionally, the following eigenvalue solutions solve for the neutron flux as $t \to \infty$; in other words, the eigenvalue solutions represent the asymptotic behaviour of the flux with respect to time. This asymptotic behaviour corresponds to the *fundamental mode* eigenvalue, and higher mode eigenvalues of the neutron flux are assumed to be transitive, and thus are also disregarded [9].

2.2.2.1 The multiplication eigenvalue

This method is often referred to as the k-eigenvalue, λ-eigenvalue or k-static method for solving criticality problems. To balance Equation 2.10, it is written as an eigenvalue problem. That is,

$$(T - S)\,\psi \;= \lambda F \psi \tag{2.11}$$

where $\lambda = 1/k$ [9]. As mentioned above, only the eigenvalue corresponding to the fundamental mode is to be considered, which is the largest value of k that solves Equation 2.11. Note that the multiplication eigenvalue k (or more correctly $1/k$) found by solving Equation 2.11 is not necessarily the neutron multiplication constant, k_{eff}, but in many cases they are equivalent. This discrepancy arises from the assumed time-independence and will be discussed in Section 2.2.2.3.

The addition of a constant $1/k$ to Equation 2.10 is generally thought of as adjusting the neutron yield per fission, $\nu\Sigma_f^i$, so that the left and right sides of the equation balance (i.e. $\nu \to \nu/k$).

$$(T - S)\,\psi \;=\; \chi_p(E) \sum_i \left(1 - \beta^i\right) \int dE' \int d\Omega' \, \frac{\nu}{\mathbf{k}} \Sigma_f^i \psi \tag{2.12}$$

So if the system is subcritical, $k < 1$ and $\nu/k > \nu$; the introduction of k increases the neutrons per fission and balances the equation making the *new* system critical. Recall that Equation 2.10 was *modified*, so Equation 2.11 does not solve the actual system, rather it is solving an idealized critical system that was created when k was introduced [10]. It is the difference between these two systems $(1/k)$ that gives a quantitative measure of how far the real system is from critical.

2.2.2.2 The time-absorption eigenvalue

Instead of simply adding a scalable eigenvalue to Equation 2.10, the time-absorption eigenvalue method of solution assumes that the problem is separable and that the time-dependent part is an exponential [9]. That is

$$\psi(\overrightarrow{r}, \hat{\Omega}, E, t) = \psi_\alpha(\overrightarrow{r}, \hat{\Omega}, E)\, e^{\alpha t} \tag{2.13}$$

Then Equation 2.7 becomes

$$(T - S - F)\,\psi = -\frac{\alpha}{\upsilon}\,\psi \tag{2.14}$$

by taking the derivative of Equation 2.13 with respect to time. Equation 2.14 is an eigenvalue equation in terms of α, so this solution method is often referred to as the α-eigenvalue method (also referred as the ω-eigenvalue method). Again, the fundamental mode solution corresponds to the (real part of) the largest value for α that solves Equation 2.14. The α eigenvalue is called the time-absorption eigenvalue because it essentially adds an absorption (loss) term to the equation that comes from the time-dependent part of the flux. However, in a subcritical medium, α is negative, so the eigenvalue term becomes a neutron source. Additionally, the extra absorption term depends on the inverse of the speed (kinetic energy) of the neutrons. Therefore, the eigenvalue preferentially removes low energy neutrons in supercritical systems, and preferentially adds low energy neutrons in subcritical systems [9].

2.2.2.3 Comparison of eigenvalue methods

The eigenvalue solutions described above differ in their treatment of the time-dependent part of Equation 2.5. The multiplication eigenvalue eliminates time-dependence by assuming that the neutron flux is time-independent. Thus, any solution of the k eigenvalue for an unstable system will be inaccurate because the system fails to meet the fundamental assumption made in Equation 2.11 (i.e. the flux is *time-independent*) [10]. Conversely, the α eigenvalue solution does not require the system to be time-independent, but the time-dependence of the neutron flux is restricted to exponential growth or decay controlled by the fundamental eigenvalue, α. The α eigenvalue solution should not be used if the system properties undergo quick changes, since these can evoke higher order modes (eigenvalues) where the change in the neutron flux can no longer be modelled by a simple exponential.

However, when the basic conditions of each solution method are met, they are both valid solutions of Equation 2.5. The eigenvalue solutions from both methods can be compared using

$$k_\alpha = \alpha T_R + 1 \tag{2.15}$$

where k_α is the neutron multiplication constant corresponding to the α eigenvalue, and T_R is the average neutron removal time [10]. The neutron removal time is defined as the time from the creation of a neutron (through fission, etc) until it is removed from the system by absorption or by escaping the system geometry. Obviously, this comparison is only valid when the preconditions of both solution methods are met.

2.2.3 Dynamic versus static criticality

Dynamic criticality refers to a direct calculation of the neutron production and loss rates (see Equation 2.8). This is usually achieved by calculating the total neutrons produced and lost over a time period, T, such that

$$k_{eff} = \frac{P(t) \times T}{L(t) \times T} = \frac{N_P(T)}{N_L(T)} \tag{2.16}$$

where N_P and N_L are the total neutrons produced and lost over T, respectively [10]. Equation 2.16 is true as long as the geometry and materials of the simulation world remain constant over T. Therefore, this definition of criticality is valid for sub- and supercritical systems [10].

Static criticality refers to criticality solutions that are time-independent. The eigenvalue solution methods listed above are both static solutions, although the α-eigenvalue method does assume an exponential time dependence. Other than the α-eigenvalue method, static criticality solution methods can only (accurately) solve for k_{eff} in a (near) critical system [10]. Another common static criticality method is to define the criticality of a system by comparing the number of neutrons in successive fission generations. That is

$$k_{gen} = \frac{\text{Number of neutrons in generation } i+1}{\text{Number of neutrons in generation } i} \tag{2.17}$$

where neutrons born through fission form the next generation; neutrons born through other interactions do not advance the *fission* generation of the neutrons [4]. In a real system, the neutrons in a single generation will not "die" or "spawn" new neutrons at the same time. In the time taken for slow neutrons to give birth to the next generation,

more energetic neutrons could go through multiple generations. Therefore, calculating k_{gen} is inherently time-independent, and is only valid for critical systems (assuming the neutron population covers a range of energies).

As with the comparison of the eigenvalue methods, time-independent, or static, criticality methods may be applied to non-critical systems. In these cases, the actual value of k_{eff} will be inaccurate, but its value relative to unity can be used as a qualitative assessment of the system. In other words, a static criticality calculation will show whether the system is sub- or supercritical, and give a rough estimate of the system's difference from a critical system. Conversely, dynamic criticality calculations are always applicable, but they are often more difficult to implement than the static methods and they are unnecessary when simulating a critical system (e.g. a nuclear reactor in equilibrium conditions).

2.3 Nuclear Data

The nuclear data used in simulation codes contains the parameters necessary to model neutron interactions, most of which are indexed by the incoming energy of the particle that initiates the interaction (e.g. neutron). These parameters include interaction cross sections, outgoing angular and energy distributions, and secondary particle yields. Since experimental measurements only cover portions of the nuclear data, and different experiments may disagree, the nuclear data used in simulations comes from evaluated data sets [11]. These evaluated data sets are formed through a combination of experimental data, nuclear interaction models, and evaluator judgement. The evaluators must use the tabulated error for the experimental data and decide how closely the models should fit the given data. While these data sets are validated

against a range of characteristic simulations, the data sets do contain inconsistencies relative to the true physical parameters and to other evaluations of the same data.

Standard nuclear data sets are produced in series as the evaluations are revised and updated. Some popular evaluation series are the ENDF/B series produced by Brookhaven National Laboratory, the ENDL series produced by Lawerence Livermore National Laboratory, and the JENDL series produced by the Japan Atomic Energy Agency. In this thesis, the ENDF/B series is used exclusively.

2.3.1 Data Formats

Most simulation codes require a specialized format for the nuclear data regardless of the source evaluation [4, 12]. Simulation codes such as MCNP[1] and Geant4 use ENDF/B data, but the data is reformatted so that it can be read directly by the simulation code (without parsing the original ENDF/B file). This reduces the initial computational overhead required to start a simulation, but it also reduces the portability of data sets between codes; a data set created for MCNP cannot be used directly in Geant4.

MCNP

The MCNP data format is standardized as three tables of data. The first lists the totals pertaining to the different data types, such as the total number of energy data points or the total number of interactions that produce secondary neutrons. The second contains indices of each different data type in the third table. The third table contains the actual nuclear data including incoming neutron energies, interaction cross sections, and outgoing angular

[1]Monte Carlo N-Particle is a stochastic simulation code that models the transport of neutrons, photons and electrons. It was developed by Los Alamos National Labs (LANL) [4].

and energy distributions. This data format is compact, but is hard to read without first parsing the data according to the indices listed in the second table. The data library format is given in the third volume of the MCNP manual [4].

Geant4 - G4NDL

The neutron interaction data in Geant4 for neutrons in the range of [0,20 MeV] is given by the Geant4 Neutron Data Library (G4NDL). The data is separated into interaction cross sections and final state data (i.e. the state of all constituent particles at the end of an interaction). These two data types are further divided into elastic scattering, radiative capture, fission, and all other inelastic interactions. The final state data includes all relevant parameters for each interaction (outgoing angular and energy distributions, secondary particle yields, etc.) but this data is read directly and sequentially by the code without any parsing, so it is also difficult to read. Unfortunately, the G4NDL libraries do not come with a format manual, so the format must be deciphered by reverse-engineering it from the source code [12].

2.3.2 Doppler Broadening of Resonances

In general, the low (thermal) and high energy (> 1 MeV) cross section data is relatively smooth, but some interactions contain resonant peaks in the middle energies (approximately 1 to 100 keV). These peaks are narrow, only covering small energy regions, but they may be several orders of magnitude in height. Thus, for a small localized energy region, the interaction cross section can be many times greater than that of the surrounding energies, and may even be greater than the cross sections at

the thermal energies. This so-called resonance region contains many such peaks (see Figure 2.1), and is very important in reactor calculations. At a resonance peak, a neutron-nucleus interaction is likely to form a compound nucleus

$$n_0^1 + X_Z^A = X_Z^{A+1} \tag{2.18}$$

if the energy of the neutron and nucleus is close to the energy of the compound nucleus minus the binding energy of the neutron. Outside of this energy range, the probability of forming a compound nucleus is lower, so the interaction cross section is also lower, which forms a peak in the cross section profile [1]. These compound nuclei, and associated resonances, occur for fission, radiative capture, and certain scattering interactions.

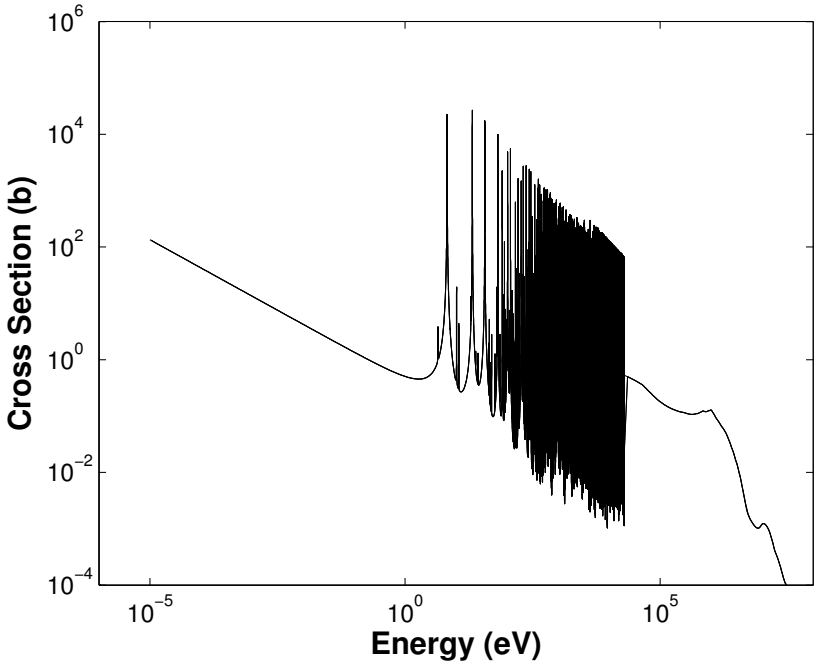


Figure 2.1: Radiative capture cross section of Uranium-238 exhibiting resonance peaks [13].

The nuclear cross section data is evaluated at a single temperature (often 0 K), and must be re-evaluated at higher temperatures for most simulations. This reevaluation is necessary because at any temperature above 0 K, the nucleus is moving (vibrating). Thus, the incoming neutron velocity in the direction of travel of the nucleus is given by

$$\overrightarrow{v}' = \overrightarrow{v} - \overrightarrow{V} \tag{2.19}$$

in the rest frame of the nucleus, where $\overrightarrow{v}$ is the neutron velocity in the lab frame and $\overrightarrow{V}$ is the current velocity of the nucleus due to vibrations (assuming non-relativistic speeds since the neutron energy is relatively low) [1]. The velocity of the nucleus can be approximated by a Maxwell-Boltzmann distribution describing an ideal gas in equilibrium at a temperature T. Therefore, the speed of the nucleus is approximately $V_{th} = (kT/M)^{1/2}$, so at higher temperatures, the nucleus is moving with greater energy in a random direction [1]. This effectively broadens the resonant peaks because neutrons that have energies above or below a resonant energy in the lab frame, may now have the proper energy to form a compound nucleus in the rest frame of the nucleus, and vice versa for neutrons already at the resonant energy. The process of broadening the resonance peaks to account for the temperature of a material is known as Doppler broadening, and is shown in Figure 2.2.

Doppler broadening may be applied to a data library before it is used in a simulation using nuclear data utility codes such as NJOY [15]. However, some simulations use on-the-fly Doppler broadening [16]. In this case, the following algorithm is used to determine the cross section at a temperature T.

Algorithm 2.1: On-flight Doppler broadening

Input: Incident neutron, target nucleus, material temperature T, convergence limit $\delta = 3\%$, $\sigma_{sum} = 0$

Output: Interaction cross section at T

$N = \max(10, T/60)$ `// Number of nucleus velocities to sample`

Repeat

- $\bar{\sigma} = \sigma_{sum} \,/\, count$ `// Save last average cross section`
- **for** $i = 1 \rightarrow N$ **do**
  - $\overrightarrow{V} = r\,(kT/M)^{1/2}$ `// Sample nucleus velocity`
  - $\overrightarrow{v}' = \overrightarrow{v} - \overrightarrow{V}$ `// Find rest frame velocity`
  - $\sigma_i = \sigma\left(\frac{1}{2}mv'^2\right)$ `// Find cross section at` v'
  - $\sigma_{sum} = \sigma_{sum} + \sigma_i$ `// Add cross section to sum`
- **end**
- $count = count + N$ `// Update counter`

Until $(\bar{\sigma} \;<\; (1 \pm \delta)\,\sigma_{sum} \,/\, count)$

$\bar{\sigma} = \sigma_{sum} \,/\, count$

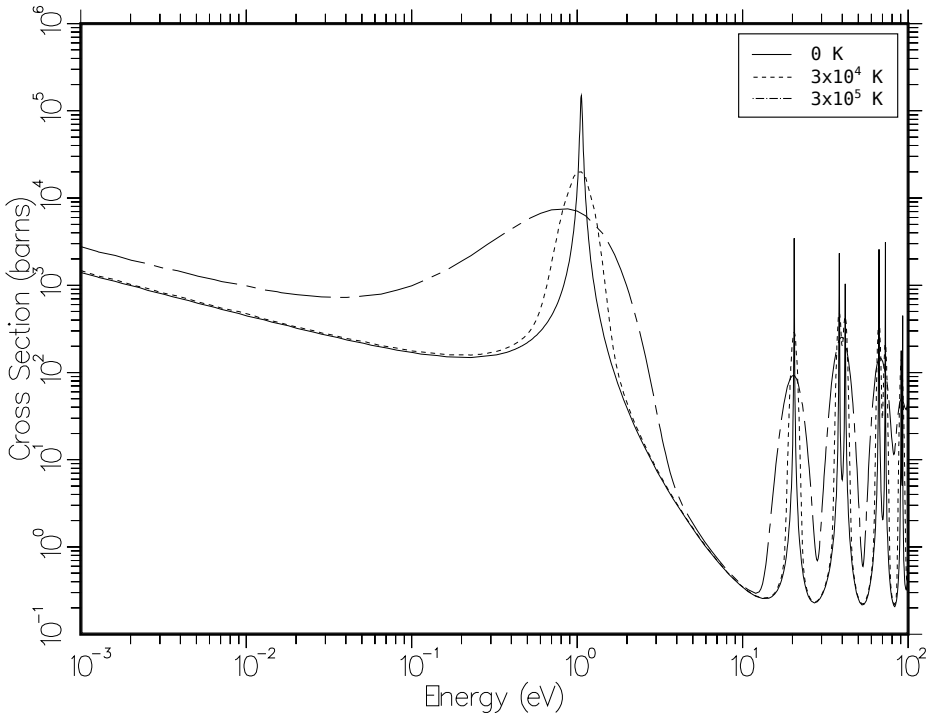


Figure 2.2: Broadened resonances in radiative capture cross section of Pu-240 due to temperature [14].

where r is a Gaussian-distributed random number centred at one [12]. The on-flight Doppler broadening algorithm samples the velocity of the nucleus and determines the neutron interaction cross section in the rest frame of the sampled nucleus. This process continues until the average cross section value converges within the given limit. For efficiency, the sampled cross sections are calculated N times before the convergence condition is re-evaluated. Unfortunately, this algorithm becomes increasingly inefficient as the temperature of the simulated material increases relative to the temperature at which the nuclear data was evaluated. Therefore, a nuclear data library that was evaluated at a temperature near the simulated material temperature is preferred.

2.3.3 Energy Discretization

In general, the nuclear data evaluations are continuous in the energy range of the evaluation. While the evaluation itself may be in the form of a table of discrete points, the evaluation includes sufficient data points such that accurate data may be obtained at any energy in the given range through interpolation. However, for some simulation codes, especially deterministic codes (see Section 2.4), calculations using continuous energy are too complicated and costly in terms of computation. Therefore, the energy range is discretized into energy groups, where each energy group has a single interaction cross section, as well as singular energy and angular distributions for secondary particles [1]. However, since the neutrons may lose or gain energy, these *multigroup* data libraries must also contain the up- and down-scattering cross sections that provide the probability of a neutron being scattered up or down to a particular energy group [1].

Since several resonance peaks may be contained within a single energy group, the averaged interaction cross section does not accurately reflect the effects of these resonances. In particular, the resonances cause local depressions in the flux distributions at the resonant energies; this effect is known as *self-shielding* [9]. Therefore, further self-shielding corrections are needed to accurately model materials that contain resonance peaks.

2.4 Deterministic Simulations

Deterministic simulations are labelled as such because their solutions do not contain any randomized elements. Therefore, the solution of a deterministic simulation will

not change unless the problem parameters are changed, and thus, the solution is predetermined. In general, deterministic solutions of the neutron transport problems are faster than stochastic solutions, but integral to this rapidity are several simplifications that make Equation 2.5 manageable. Additionally, deterministic problems are solved numerically since even simple solutions of the neutron transport equation are difficult to obtain analytically [1].

2.4.1 Simplifications

Three major simplifications are applied to the simulation for most deterministic solvers. The simplifications are as follows

Energy Discretization

The first simplification used in most deterministic nuclear codes is the energy discretization described in Section 2.3.3. With this simplification, the transport equation becomes a set of linear equations for each energy group and movement between the energy groups is handled by group-specific scattering cross sections (e.g. one scattering cross section would define scattering from energy group one to energy group two).

Spatial Discretization

The second simplification is to discretize the simulation geometry in space, such that the simulation world is composed of N_V volumes with homogeneous material properties. This simplification is generally necessary to solve the neutron transport equation numerically within the simulation geometry [17].

Angular Discretization

The third simplification is the discretization of the outgoing angular distributions of the hadronic interactions. Thus, the angular distributions are reduced to a set of directions, $\hat{\Omega}_n$, where $n = 1, ..., N$ [1]. Generally, these directions should be chosen to be normal to the surfaces (edges) created by the spatial discretization; this makes surface currents easier to calculate.

2.4.2 Static Solutions

The most common method for solving the neutron transport equation requires two steps. The first is to solve the integral form of the transport equation for a small, repeatable geometry (i.e. a cell) to obtain homogenized cross sections. These homogeninized cross sections can be applied across the entire cell so that the simulation world becomes a series of cells that each have a single set of interaction cross sections for each energy group [17]. The second step is to iteratively solve for the neutron flux between the cells. This second step, the diffusion approximation, is not strictly necessary, but by combining these two steps, a complex problem, such as a nuclear reactor with instrumentation and control rods, can be reduced to a simpler, more easily solvable problem.

2.4.2.1 Cell Homogenization

The first step is to solve the neutron transport equation over the cells using the three simplifications listed above. This can be done using deterministic numerical simulation codes such as DRAGON. DRAGON specifically uses the *collision probability method* to solve for the absorption and scattering cross sections (Σ_a and Σ_s), the fission yield ($\nu\Sigma_f$), the fission spectrum (χ), and the angular neutron flux (ϕ) in each

volume of the cell [18]. Then the homogenization for each energy group g is performed by averaging each parameter across the volumes. Homogenizing these parameters for every energy group yields a compressed data library where each cell only has one value for each property and energy group.

2.4.2.2 Diffusion Approximation

Once the cells have been homogenized, the diffusion approximation is used to calculate the flux between cells. First an initial flux is set for each cell, and boundary conditions are set for cells on the edge of the system. Then the system is solved using a finite difference method to determine the flux at the centre of each cell. Since the cells are homogeneous and the cross sections are known, the unknown quantity is the flux into the cell from adjacent cells. By conservation of particles, the flux into cell A from cell B must be equal to the flux out of cell B into cell A. Therefore, the following algorithm is used to converge to the actual flux shape

Algorithm 2.2: Iterative flux solution

Input: Homogenized cross sections, initial flux values ϕ_0, convergence limit δ

Output: Final flux profile

/* Keep recalculating the neutron flux at the centre of each cell till they all converge within δ */

Repeat

  /* Find flux at the centre of each cell in the system */

  for $j = 1 \rightarrow N_V$ **do**

    Find ϕ_i^j based on adjacent cells // ith flux of cell j

  end

  $i = i + 1$ // Increment i

Until $\max\left[(\phi_{i+1} - \phi_i)/\phi_{i+1}\right] < \delta$

As the calculated flux at the centre of cell j is updated, this causes the fluxes of all adjacent cells to change; hence, continuing the iterative process. This process (or a similar one) can be completed by codes such as DONJON [18].

2.4.3 Dynamic Solutions

In general, time-dependent deterministic solutions of the neutron transport equation either use the *point kinetics approximation* or a more complicated space-energy dynamics approach. The first approach, the point kinetics approximation, assumes that

the neutron flux shape is time-independent. That is

$$\psi(r, E, t) = p(t)\,\phi(r, E) \tag{2.20}$$

where $p(t)$ is a time-dependent amplification factor and ϕ is the time-independent flux shape. This assumption leads to the point kinetics equations, which are given by

$$\begin{aligned} \frac{dp}{dt} &= \frac{\rho(t) - \beta}{\Lambda(t)} p(t) + \sum_{i=1}^{n} \lambda_i C_i(t) \\ &= \frac{\rho(t) - \beta}{\Lambda(t)} p(t) + \frac{S_d(t)}{\Lambda(t)} \\ \frac{dC_i}{dt} &= \beta_i \frac{\rho(t)}{\Lambda(t)} p(t) - \lambda_i C_i(t) \quad | \quad i = 1, ..., n \end{aligned} \tag{2.21}$$

where ρ is the reactivity, Λ is the average neutron generation time, i is the delayed neutron precursor group number, n is the total number of precursor groups, β is the total delayed neutron fraction, β_i is the delayed neutron fraction of group i, λ_i is the delayed neutron decay constant, and C_i is the delayed neutron precursor concentration [2].

The point kinetics approximation is limited by the lack of any time-dependent spatial or energy effects. Consequently, the *quasistatic method* was developed to allow indirect time-dependence in the neutron flux shape. The flux shape in the quasistatic method is solved at regular time steps using the following equation

$$\left(F + S - T - \frac{1}{v}\frac{\dot{p}}{p}\right)\psi(r, E, t) = -\frac{1}{p} S_d\left[p(t')\psi(r, E, t')\right] \tag{2.22}$$

where F is the prompt neutron fission source, $\dot{p}$ is the derivative of the amplitude function with respect to time, S_d is the delayed neutron source term, and t' is the

time of the previous time step [2]. Note that the time step used in Equation 2.22 can be longer than the time step used to solve the point kinetics approximation. The addition of time-dependence to the neutron flux shape allows this method to be used to simulate many accident scenarios, such as a rod ejection [2].

Instead of using the point kinetics approximation, the transport equation may be solved numerically using one of the space-energy dynamics approaches. These approaches include the finite difference method, the modal and nodal approaches, and the quasistatic correction described above. The finite difference method involves replacing the differential terms in Equation 2.5 with finite difference quotients, and then the neutron flux is solved across a space-energy mesh at each time-step. The modal approach represents the flux as a superposition of fundamental space-energy modes, where the time-dependent superposition coefficients are updated at each time step. Lastly, the nodal approach represents the neutron flux as individual coupled fluxes at spatial nodes. The nodal fluxes again feature time-dependent coefficients. All of these methods are more accurate than the point kinetics approximation, but this accuracy increases the computational cost of each calculation. Moreover, the accuracy of each approach is limited by the number of fundamental divisions employed (i.e. number of meshes, modes or nodes).

2.4.3.1 Limitations

The point kinetics approximation has a number of limitations. First, Equation 2.21 does not account for energy effects. For example, the delayed neutrons are generally born at lower energies than prompt neutrons, so they tend to undergo less moderation and are absorbed more quickly. Second, the delayed neutron characteristics vary for

different fissile isotopes, so the β_i and λ_i values must be calculated as an average of the individual values for each isotope. Finally, the major limitation of the point kinetics approximation is the assumption that the actual spatial flux distribution can be accurately represented using the static shape factor, ϕ. The assumption of a stable spatial shape factor is only true if the system's material composition does not change. For accident scenarios that are spatially dependent, such as a rod ejection or rod drop, the point kinetics approximation tends to underestimate the positive reactivity insertion and overestimate the negative reactivity insertion [2]. Thus, this method is not conservative, which is problematic when simulating reactor accidents.

For changing spatial distributions, quasistatic methods may be employed to achieve some time dependence in the spatial form factor. Obviously, the frequency of these periodic spatial calculations must be sufficiently large relative to the changes in the core composition; otherwise, the point kinetics approximation will diverge from the modelled system as the true spatial distribution diverges from the assumed spatial form factor [1]. Finally, the other space-energy approaches have the same limitations as other deterministic equations; the discretization of space and energy, as well as time, affects the accuracy and speed of the calculations.

2.5 Monte Carlo Simulations

Unlike deterministic calculations, which solve for values that are predetermined when the problem is posed, stochastic calculations rely on random sampling to model the same problem. In general, a problem can be separated into individual components whose behaviour can be described by probability trees; although, these probabilities

are not necessarily independent. Starting from a given initial state for the components, the stochastic calculation randomly samples the probability trees to produce a single outcome of the initial state. Assuming the random sampling process is not biased, subsequent simulations of the problem will produce different outcomes. If an infinite number of stochastic calculations were performed, the collective outcomes would exactly reproduce the combined probability trees.

Monte Carlo processes are a subset of stochastic calculations and can take a variety of differing forms, but in general, they all follow the basic outline described above. The outcome of a system is modelled by sampling probability distributions that describe the behaviour of a system given an initial state. Random outcomes are generated until the average outcome converges to within an appropriate variation. At this point, the Monte Carlo calculation should reproduce the results of the deterministic equation with limited statistical error.

The benefit of using a Monte Carlo calculation is that complex problems, like neutron transport in a material, can be modelled with relatively simple calculations. While deterministic solutions require simplifications such as energy discretization, a Monte Carlo solution needs only to have sufficient probability tables to model any possible interactions. However, this completeness, and the necessity of simulating a "complete" set of outcomes, requires longer computation times relative to equivalent deterministic calculations.

2.5.1 Basic Principles

The procedure of random sampling to determine the outcome of a problem is often referred to colloquially as "rolling the dice". To choose a particular outcome from a

table of probabilities, a random number generator produces a number, r, and uses

$$\exists\, m \in [1, N] \quad | \quad \frac{r}{r_{max}} \sum_{i=1}^{N} P_i \leq \sum_{i=1}^{m} P_i \tag{2.23}$$

where r_{max} is the maximum number that the random number generator can produce, N is the total number of outcomes, and P_i is the probability of outcome i occurring. This is analogous to assigning outcomes to the numbers on a die, and then rolling the die to produce a random result. In probability theory, the set of all possible outcomes, S, is known as the sample space, and any subset of S is referred to as an event, E [19]. As the number of calculated outcomes increase, the Monte Carlo solution approaches the actual solution (the particular combination of the probability tables that the problem represents). This process of solving problems by simulating and counting outcomes is used in neutron transport to measure quantities such as aggregate cross sections and criticality (see Equations 2.3 and 2.8).

The following discussion will focus mainly on neutrons and neutron interactions. However, most of the following is equally applicable to any other particle that the Monte Carlo code can simulate including protons, electrons, alpha particles, etc.

2.5.2 Neutron Transport in Monte Carlo Simulations

Monte Carlo simulations are used to solve Equation 2.5 by converting the integral production and loss terms into probabilities, and applying these probabilities to a neutron moving through a material. The neutrons are followed in space from creation to loss, where creation occurs from a neutron-producing interaction (e.g. fission, original source particle creation), and loss can occur through a neutron-removing interaction (e.g. absorption, escape). Along this path, the neutrons move through

a series of *steps*, each of which ends with an interaction. The length of each step and the type of interaction are calculated using the interaction cross sections with Equations 2.1 and 2.4. Additionally, secondary neutrons, which are produced from interactions such as fission, are also followed until their loss. Starting with a single neutron, the result of a Monte Carlo simulation is the outcome of that neutron and all of its descendants; that is, the combined history of all the neutrons originating from the first neutron, which includes the number, location and timing of all interactions, secondary neutron productions and neutron losses. In general, an *event* is taken to be the history of n initial neutrons and their descendants from creation to loss.

In most Monte Carlo transport codes, a neutron is only subject to external forces during the interactions at the end of each step. Between the beginning and end of a step, the neutron moves along a straight path with a constant velocity [4]. Additionally, the interactions are instantaneous so a neutron can be assumed to travel the entire step length without any deviations from its initial velocity. All of these conventions derive from the assumptions listed in Section 2.1.3.

2.5.2.1 Step-by-Step Transportation

While the exact mechanics of neutron transport in Monte Carlo simulations can differ, most methods are similar to the procedure described below. The following procedure is used to model the movement of a neutron in the simulation world through a series of steps. While this algorithm is general, it is closest to the neutron transportation algorithms used in MCNP [4].

Algorithm 2.3: Monte Carlo step-by-step transportation

Input: Nuclear data, current state of neutron

Output: Step length, neutron interaction, end state of neutron

while *neutron is alive* **do**

  /* Find new position */

  $l = -\frac{1}{\Sigma_t} \log \xi \quad | \quad \xi \epsilon [0,1)$ // ***Calculate step length

  $\overrightarrow{x} = \overrightarrow{x} + l \cdot \hat{x}$ // Calculate new position

  /* Pick the interaction type */

  $r = x\, \epsilon\, [0,1]$ // Generate random number

  $\Sigma_R = 0$ // Initialize the running total

  for $i = 1 \rightarrow TotalInteractions$ **do**

    $\Sigma_R = \Sigma_R + \Sigma i$ // Add Σ_i to running total

    if $r \leq \Sigma_R / \Sigma_t$ **then**

      break

    end

  end

  /* Model interaction i */

  if *necessary* **then**

    Kill neutron

    Create secondary particles

  end

end

where $TotalInteractions$ is the total number of neutron interactions defined for a neutron in the given material, and Σ_R is the running total of the individual macroscopic cross sections, which starts at zero for each iteration of the while loop. Stochasticity is added to the process through the random numbers ξ and r, which randomize the step length and the interaction selection respectively [4].

When the neutron crosses from one volume into the next over the course of a single step, the tracking algorithm listed above must be modified to account for the changing macroscopic cross section. However, only the step length calculation (marked with *** in Algorithm 2.3) is affected. The step length calculation becomes [9]

Algorithm 2.4: Stepping across volumes

$\lambda = \log \xi \quad | \quad \xi \, \epsilon \, [0, 1)$ // Mean path lengths left
Repeat
  $\overrightarrow{d} = Intersect(S, \overrightarrow{x}, \hat{p})$ // Intersection with surface S
  $d_{out} = \overrightarrow{d} - \overrightarrow{x}$ // Calculate distance to surface
  $l' = -\frac{1}{\Sigma_t^m}\lambda$ // Calculate step length in material m
  if $l' > d_{out}$ **then**
    $l = l + d_{out}$ // Add incremental distance
    $\lambda = \lambda - d_{out}/\Sigma_t^m$ // Reduce mean free path lengths left
  end
Until $(l' < d_{out})$

where the $Intersect$ function determines the intersection between the surface of the volume, S, and the path of the neutron, which is calculated using the position ($\overrightarrow{x}$) and momentum direction ($\hat{p}$) of the neutron. Then the interaction type and result depend on the final material, m'.

For most Monte Carlo codes, the tracking algorithm does not consider time. Instead, the time is updated at the end of each step using [16]

$$t_f = t_0 + \frac{l}{v} \tag{2.24}$$

where t_0 and t_f are the times at the beginning and end of the step respectively, and v is the scalar velocity of the tracked neutron. This assumes that the neutron does not undergo any acceleration during the step. When the tracked particles are being accelerated, such as charged particles in an electromagnetic field, neither Algorithm 2.3 nor Equation 2.24 are valid. The algorithms to deal with this situation are discussed in Section 2.6.2.

2.5.3 The Simulation World

In Monte Carlo simulations, the simulation world is composed of elementary connected regions, often referred to as volumes, and each volume is composed of a single homogenous material [9]. The simulation world is bounded by a single *mother volume*, often called the world volume. If a neutron leaves the world volume, it is *lost* and is killed (removed from the simulation). Inside the world volume, the *daughter volumes* are arranged so that their boundaries do not overlap with each other; that is, if two daughter volumes are intersecting, one of these volumes must be completely contained within the other; a neutron must be able to unambiguously determine which volume it is in so that the tracking algorithm (e.g. Algorithm 2.3) may apply the correct nuclear data.

Each elementary volume can be further divided into several smaller volumes to provide higher resolution for scoring, such as flux tallies (see section 2.5.5). The

quantization of a volume is only limited by the minimum dimension that the simulation code can resolve. However, complicating the geometry by subdividing the elementary volumes will reduce the efficiency of the tracking algorithm. As the number of volumes increases, determining when a neutron reaches the edge of its current volume and which volume it will enter next becomes increasingly difficult and time consuming.

2.5.4 Initial Source

Since Monte Carlo simulations follow individual particles in the simulation world, the simulation must begin with a set of initial neutrons (*primary particles* or *primaries*). These particles can start with any distribution of position and momentum throughout the simulation world, although certain distributions will be better suited to specific applications. The neutron distribution will converge in space and energy to the "physical" distribution dictated by the simulation world; this process is referred to as source convergence. Even subcritical simulation worlds will have an innate source distribution as long as there are sufficient neutrons in the world [4]. Below is a list of common source distributions.

1. **Beamline**

   Primary neutrons that start at the same position with the same momentum direction. This distribution is useful for probing a material or apparatus with neutrons over a range of energies.

2. **Point Source**

   Primary neutrons that start at the same position with random momentum directions. This source is simple, easy to implement, and useful for many applications. However, the neutron spatial distribution may take more time to converge.

3. **Uniform Source**

   Primary neutrons that start at locations uniformly distributed across the simulation world. This source can be used if the evolution of a point source would be too slow.

4. **Distributed Source**

   Primary neutrons whose position and/or momentum is assigned according to a given distribution. This source is the quickest to converge towards the true physical spatial distribution if the given distribution is well chosen.

In criticality simulations, source convergence is particularly important because k_{eff} is calculated using the rates of neutron production and loss in Equation 2.8. These rates will change based on the spatial distribution of the neutrons, especially if the simulation is initialized from a point source [4]. Thus, the source must be allowed to converge in a Monte Carlo criticality calculation before any criticality estimates can be made.

2.5.5 Tallies and Scoring

The results from Monte Carlo simulations come from various counting processes referred to as tallying and scoring. Particular events are noted and counted over the

course of a simulation using a running tally. For example, the neutron flux through a given volume is measured by tallying the number of neutrons entering and leaving the volume during the simulation. Individual events may also be recorded directly for further accuracy, such as the location of each fission in a simulation. Certain tallies, such as energy deposition or neutron flux, are less useful either when applied to an entire elementary volume or saved as individual events (i.e. individually recording energy deposition of each fission). For these cases, the elementary volumes are subdivided into smaller volumes (see Section 2.5.3), which are used to separate, or *bin*, the tallies into smaller, more descriptive values. This can be used to determine the energy deposited in different parts of a detector or the flux profile along the surface of a subdivided volume.

Since a Monte Carlo simulation is a stochastic process, the number of initial neutrons is directly proportional to the statistical accuracy of the tallies and the derived quantities. If the process of interest is relatively rare, such as (n,2n) collisions in Uranium-235, the number of initial neutrons must be increased to get a sufficient number of (n,2n) collisions so that the (n,2n) tally is large enough to be statistically relevant (i.e. not dominated by uncertainty). The scoring is ultimately a counting process, or formally, a Poisson process, where the number of tallied occurrences is distributed according to a Poisson distribution. Thus, if N tallies are made, then the standard deviation in this tally decreases by $\sqrt{N}$ [19]. However, the statistical error in the tally does not account for any systematic errors due to errors and approximations in the simulation code or nuclear data [4]. The expected value of a tally x can be written as

$$\bar{x} = E(x) + \delta_{system} \tag{2.25}$$

where $\bar{x}$ is the "true" mean value of the tally in a physical system, $E(x)$ is the expected value of the tally derived from the Monte Carlo simulation, and δ_{system} is the systematic error of the Monte Carlo code. The expected value will have a standard deviation, $\sigma(x) \approx 1/\sqrt{N}$, that may be reduced by increasing the number of initial particles, but the systematic error can only be minimized by modifying the Monte Carlo code or the nuclear data [4].

2.5.6 Quantitative Analysis

The tallies created by the scoring processes represent raw data from the simulation. After the simulation is finished, the tally data can be combined and manipulated to calculate derived quantities. The most pertinent quantities to this discussion are the calculation of the criticality of the system and of the Shannon entropy, which are discussed below.

2.5.6.1 Monte Carlo Criticality Calculations

Monte Carlo criticality calculations can be performed through a variety of methods. Three methods will be discussed here: the generational method, the α-eigenvalue method, and the dynamic method (see Section 2.2.1). For a discussion on these methods, see Section 2.2.3.

Generational method:

The criticality of a system may be calculated by comparing the number of neutrons in successive generations (Equation 2.17). This is equitable to the k-eigenvalue method since both methods find the eigenvalue k and are time-independent. In practice, the number of neutrons per generation is

calculated using Algorithm 2.5 [4].

Algorithm 2.5: Monte Carlo k eigenvalue calculation

Input: Monte Carlo simulation with a converged neutron source distribution

Output: Mean k value, $\bar{k}$

Repeat

`/* Simulation */`

Simulate one generation of neutrons starting from fission sites of the last generation

`/* Calculations */`

Tally n_{i+1}^f `// Tally number of fissions`

$k_i = n_{i+1}^f \,/\, n_i^f$ `// Calculate` k_i

$\bar{k} = \frac{1}{i}\sum_{j=1}^{i} k_j$ `// Recalculate` $\bar{k}$

$i = i + 1$ `// Increment` i

Until ($\bar{k}$ *has converged*)

For this algorithm, it is equivalent to count the number of fissions or the number of neutrons produced from fission in a generation assuming the materials are constant in time (i.e. constant neutron fission yield, ν).

Time-absorption eigenvalue method:

The time-absorption eigenvalue method uses Equation 2.15 to calculate k_α. Therefore, the Monte Carlo simulation needs to calculate the α eigenvalue

and the mean neutron removal time, T_R. If the neutron population is known at times t_1 and t_2 (where $t_2 > t_1$), then the α eigenvalue can be calculated using

$$\alpha = \frac{1}{t_2 - t_1}\left[\log\left(\frac{N(t_2)}{N(t_1)}\right)\right] \tag{2.26}$$

where $N(t)$ is the total number of neutrons in the simulation at time t [10]. The neutron population at each time is found by tallying all the neutrons in the simulation at these times, and the mean neutron removal time, T_R, can be derived by averaging the removal times of all neutrons that are killed in the interval $[t_1, t_2]$.

Dynamic criticality method:

The dynamic criticality method is implemented using Equation 2.16. The criticality of the simulation world at a time t is

$$k_{eff} = \frac{N_P(T)}{N_L(T)} \tag{2.27}$$

where T is the interval $[t_0, t]$, and t_0 is any time after the neutron source distribution has converged. Equation 2.27 is true so long as neither the geometry nor the materials of the simulation world change in the interval $[t_0, t]$.

2.5.6.2 Shannon Entropy

Shannon entropy was introduced in 1948 by Claude Shannon as a quantitative measure of the average information needed to encode a message for lossless compression [20]. The message is treated as a sequence of independent and identically distributed random variables, and the entropy of a message depends on the probability that each variable has a given value. If the message is the sequence of letters 'AABDC', then the Shannon entropy is the entropy of the set of probabilities $\{p_A, p_A, p_B, p_D, p_C\}$. Moreover, the Shannon entropy approximates the information needed to encode each letter (e.g. the average number of bits per letter needed to encode the message in a binary (digital) representation) [20].

Instead of a sequence of letters, the Shannon entropy is calculated for a discretized spatial distribution in Monte Carlo simulations, specifically the spatial distribution of the fission sites (positions of fission interactions). For a stable distribution, the Shannon entropy is constant; therefore, calculating the Shannon entropy can affirm convergence in the spatial distribution of the fission sites [4]. To calculate the Shannon entropy, the positions of the fission sites are discretized using an arbitrary three dimensional mesh. Then the Shannon entropy is

$$H_{src} = -\sum_{j=1}^{N_b} P_j \log_2 (P_j) \tag{2.28}$$

where H_{src} is the calculated Shannon entropy, N_b is the total number of grid boxes formed by the mesh, and P_j is the probability of a fission site being in box j. This probability is given by

$$P_j = \frac{n_s^j}{N_s} \tag{2.29}$$

where n_s^j is the number of fission sites in box j and N_s is the total number of fission sites [4]. If a grid box contains no fission sites, then it does not contribute to the Shannon entropy. Likewise, if all of the fission sites are contained in only one grid box, then the Shannon entropy is also zero. However, if the fission sites are uniformly distributed among the grid boxes, then the Shannon entropy will attain a maximal value of

$$H_{max} = -\log_2\left(\frac{1}{N_b}\right) \tag{2.30}$$

Given that the maximal Shannon entropy depends on the resolution of the discretization mesh, it is useful to define the Shannon entropy of a spatial distribution as a percentage of the maximal Shannon entropy, which depends on the meshing.

2.5.7 Delayed Neutrons

Delayed neutrons may be simulated in Monte Carlo simulations directly without any simplifications by stochastically simulating the decay of the fission products from each fission. However, as an approximation, the properties of the resultant delayed neutrons can be simply sampled from the existing nuclear data (e.g. multiplicity, momentum, time of birth). This approximation is simpler, but its accuracy relies on the simplifications made in the sampled nuclear data (e.g. the precursor grouping described in Section 2.1.3).

The delayed neutron decay time, l_{decay}, is determined from the difference between the time of birth of the delayed neutron and the time of the fission that produced the initial delayed neutron precursor. That is

$$l_{decay} = t_{decay} - t_{fission} \tag{2.31}$$

where t_{decay} is the time of the reaction product decay that produces the delayed neutron, and $t_{fission}$ is the time of the fission that created the initial reaction product. Thus, the simulation begins tracking a delayed neutron at

$$t_{start} = t_{fission} + l_{decay} \quad | \quad l_{decay} >> l_{prompt} \tag{2.32}$$

where l_{prompt} is the average lifetime of a prompt neutron, t_{start} is the time at the start of the tracking of the delayed neutron, and $t_{fission}$ is the time when the fission that produced the delayed neutron precursor occurred [1]. Since the delayed neutron decay time may be of the order of seconds, whereas as the lifetime of a prompt neutron is of the order of milliseconds at most, the delayed neutrons are born many prompt generations after their sister (prompt) neutrons. It is this difference that makes delayed neutrons important even though they make up less than 1% of the total neutron population. The delayed neutrons represent a much older state of the reactor, and consequently make the reactor response sluggish. This makes the reactor controllable because it gives the control systems more time to respond [1].

Given that the delayed neutrons are born much later than prompt neutrons, it can be impractical to simulate enough generations such that the delayed neutrons begin to be born in the simulation of a time-dependent neutron population (i.e. run the simulation for more than l_{decay} seconds). This challenge is often simplified by making the delayed neutrons "prompt" so they are born at $t_{fission}$ [4]. In order to make this simplification more accurate, the lifetime of the delayed neutron can be set to l_{decay} at birth, so that the average neutron generation time will reflect the presence of delayed neutrons. In general, the lifetime of a neutron is a measure of the time between the birth and death of a neutron, but to make the calculation of the average

neutron generation time easier, the delayed neutron decay time can be added to the lifetime, reducing the number of variables that need to be tracked.

2.6 Monte Carlo Implementation in Geant4

As described in Section 1.2, the Geant4 Monte Carlo Toolkit was developed in an effort to create a modern, extensible tool for Monte Carlo simulations over a wide range of applications [3]. As such, Geant4 builds on the Monte Carlo knowledge that has been gained over the past 70 years, but it also differs from older codes such as MCNP in a few key areas. The following sections will elaborate on these differences and explain how important algorithms are implemented in Geant4.

2.6.1 General Simulation Flow

Figures 2.3 and 2.4 show the organization and flow of control in Geant4. Figure 2.3 stops at the tracking algorithm that processes the tracks of an event, and Figure 2.4 continues from this point until the tracking algorithm is finished. Simulations in Geant4 are organized as a series of loops, where each action in the hierarchy completes N_i iterations of the next lower action before the higher action can finish. For example, each run will consist of N_E events that must be fully simulated before the run is allowed to end. This structure extends to the track level, but at the step level, the stepping action continues until the neutron being tracked is removed from the simulation (dies) [12]. Note that the "x++" notation is a C++ convention denoting an unary increment of the variable x (i.e. $x = x + 1$).

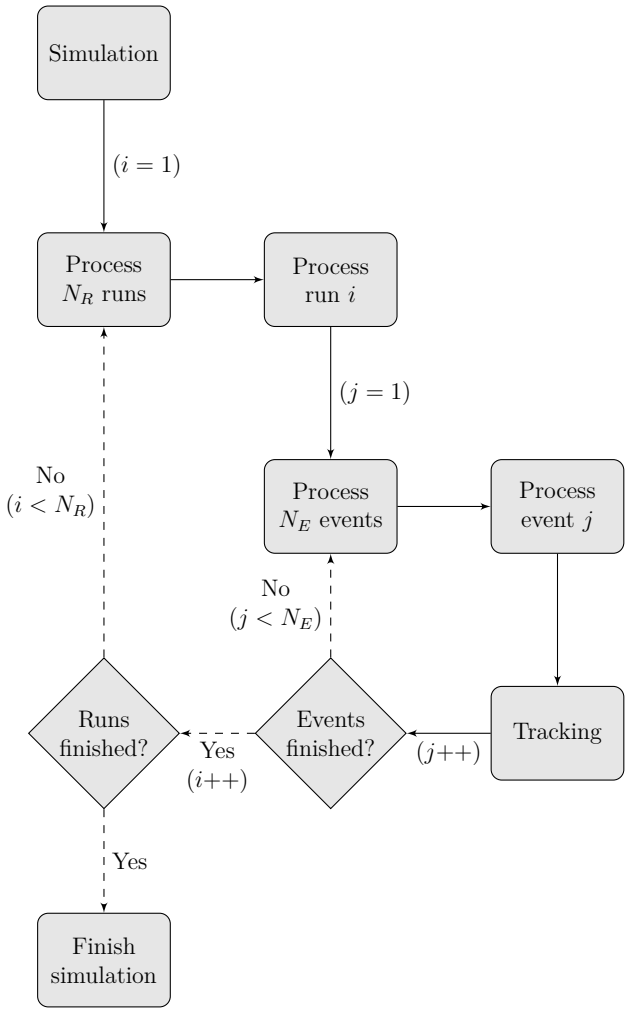


Figure 2.3: High level simulation flow diagram for Geant4

2.6.2 Transport in Geant4

As in other Monte Carlo simulation codes, Geant4 tracks particles as they move in a series of steps with instantaneous interactions at the end of each step. However, to enhance the flexibility of Geant4, any arbitrary process may be defined so long as it does the following [3]:

Declare applicability:

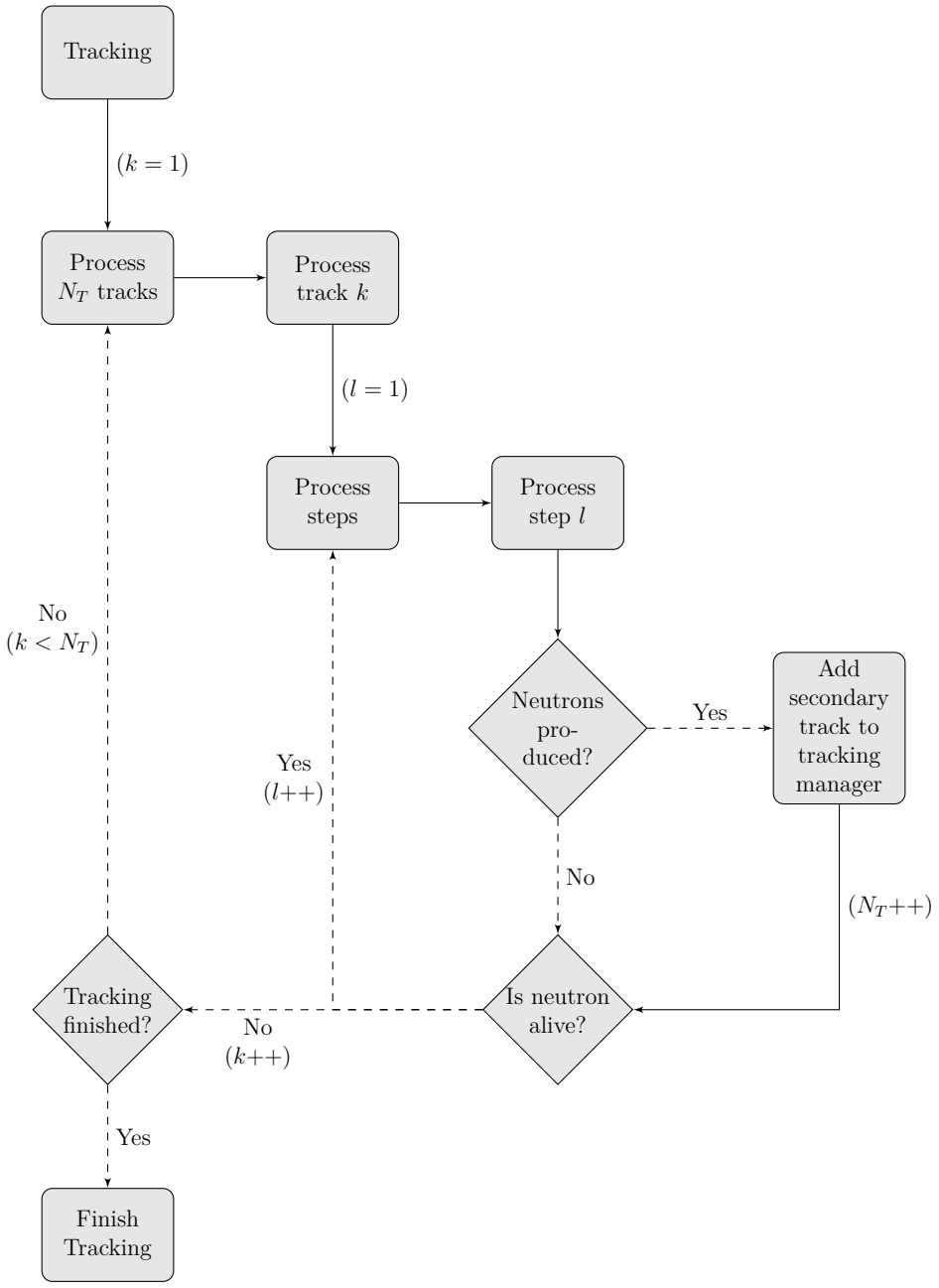


Figure 2.4: Tracking flow diagram for Geant4

Given the particle being tracked and the material being traversed, the process determines whether or not it is applicable. If not, the process is ignored

for the current step.

Propose a step length:

When prompted by the tracking manager, the process must propose a valid step length for the neutron (i.e. in the range $[0, 10^{302}]$ mm).

Model the interaction if chosen:

If the process is selected by the tracking manager, it must model its interaction by proposing changes to the attributes of the neutron (e.g. momentum, state - alive or dead, etc.). The process must also create any secondary particles that are produced in the interaction and set their attributes.

For arbitrary processes, the interaction does not have to change *any* attributes of the neutron, and in this case, the interaction will be referred to as a *null interaction*. Additionally, if a process is deemed applicable but should still not occur, the process may return the maximum step length (10^{302} mm). This could occur if the interaction cross section is zero for the current energy of the neutron. Finally, interactions that occur at the end of the step are referred to as *post-step* interactions in Geant4; Geant4 also allows *pre-step* and *along-the-step* processes. Along-the-step processes occur "continuously" along the path of the step, and will be discussed in Section 2.6.2.3.

2.6.2.1 Proposed Step Lengths

For processes that depend on cross section data, such as hadronic processes for scattering, fission and radiative capture, the proposed step length, d, is given by the following equation

$$d_i = \frac{\eta_\lambda^i}{\Sigma_i} \tag{2.33}$$

where d_i is the step length for the hadronic process i, Σ_i is the macroscopic cross section associated with i, and η_λ^i is a counter of mean free path lengths (Section 2.1.2.1) [16]. Therefore, η_λ^i is the number of mean free path lengths that the neutron has to travel before undergoing an interaction of type i. That is

$$\begin{aligned} \eta_\lambda^i &= \frac{\text{distance to next interaction of type } i}{\Sigma_i^{-1}} \\ &= -\log(r) \quad | \quad r \in (0,1] \end{aligned} \tag{2.34}$$

where r is a uniformly distributed random number between 0 and 1 [16]. In the Geant4 documentation, η_λ^i is referred to as the *number of interaction lengths left*, where an interaction length is the same as a mean free path length. Each process based on nuclear data maintains a unique η_λ^i that is initialized using Equation 2.34 at the start of tracking. After a non-null interaction, one in which the attributes of the neutron are changed, the η_λ^i value for each process is reset using Equation 2.34 [16].

For processes that do not depend on nuclear data, the proposed step length is arbitrary. The most common use of these arbitrary (in terms of step length) processes is as a *step limiter*. In this case, the proposed step length is a value d_{max} that is set according to the function of the process. For example, the transportation process is responsible for stopping neutrons at the interface of two volumes. In this case, the proposed step length is the distance to next volume in the direction of travel of the neutron [16]. That is

$$d_{max} = |\, \overrightarrow{x_S}(x_0, \hat{p}) - \overrightarrow{x_0} \,| \tag{2.35}$$

where $\overrightarrow{x_0}$ is the current position of the neutron, $\hat{p}$ is the momentum direction of the neutron, and $\overrightarrow{x_S}$ is a point on the surface of the current volume in the direction of $\hat{p}$

starting from $\overrightarrow{x_0}$.

For step limiter processes, the modelled interaction is null because the neutron is stopped arbitrarily at a point in space. Given that the step length is arbitrary and not contingent on any nuclear interaction data, the occurrence of a null interaction should not change the results of the simulation. To ensure continuity between two steps that have been arbitrarily separated by a null interaction, the η_λ^i value is not reset after a null interaction. Instead, η_λ^i is decremented by the number of interaction lengths that the neutron travelled during the step preceding the null interaction. Therefore, for any step that ends with a null interaction, the η_λ^i values are modified using

$$\eta_\lambda^i(j+1) = \eta_\lambda^i(j) - d_j \, \Sigma_i^j \tag{2.36}$$

where d_j is the step length of step j, and Σ_i^j is the macroscopic cross section of interaction i in the material the neutron travelled through during step j [16]. Thus, the effect of Algorithm 2.4 is replicated in Geant4.

2.6.2.2 Process Selection and Utilization

Since every process proposes a step length, the *winning* process is the process with the smallest proposed step length. The process selection and utilization algorithm is executed by the stepping manager as follows [12]

At the pre- and along-the-step stages, any applicable interaction will occur. Additionally, the transport of the neutron is handled by the along-the-step component of the transportation process, which may also be the winning post-step interaction if

Algorithm 2.6: Process selection and utilization in Geant4 for one step

Input: Process list, current state of neutron, current volume
Output: Final state of neutron and secondaries at the end of the step

`/* Chose winning process */`

Applicable processes each propose a step length d_i
Process with d_{min} is chosen

`/* Pre-Step */`

All applicable **pre-step** interactions occur

`/* Along-the-Step */`

All applicable **along-the-step** interactions occur
  Neutron is moved d_{min} by transportation process

`/* Post-Step */`

Winning **post-step** interaction occurs
if *interaction is null* **then**
  eta^i_λ values are decremented (Equation 2.36)
end
else
  eta^i_λ values are reset (Equation 2.34)
end

the distance to the volume surface is less than any other proposed step length [3].

2.6.2.3 Transport in an Acceleration Fields

One strength of Geant4 is its ability to model charged particles moving in an accelerating field. For example, an electron in a magnetic field will travel along a curved path. In Geant4, this is modelled by approximating the curved path as a series of short straight steps (chords) [16]. An electromagnetic process is used to limit the step to the length of a single chord, and to correct the path of the charged particle so that it follows the predicted curve. The shape of the curve itself is predicted using Runge-Kutta integration [16]. Thus, the charged particle will travel approximately η_λ^i mean free path lengths before undergoing an interaction of type i even though the path of the particle is now curved. The ability to model such events is useful when designing charged particle detectors, such as the detector systems used in particle physics experiments like the Large Hadron Collider.

2.6.3 Nuclear Data

The nuclear data in Geant4 can be in a variety of forms depending on the implementation of a particular physics process. The majority of data is stored in tables to be used either in parameterized models or directly through interpolation. The neutron high precsision (NeutronHP) physics processes used in this thesis use the G4NDL libraries (Geant4 Neutron Data Libraries), which store the interaction cross sections in tables indexed by incident neutron energies [16]. In addition to the cross section data, the G4NDL libraries contain *final state data* for each interaction; that is, the secondary particle yields, and angular and energy distributions necessary to model

a given hadronic interaction. The final state data is also stored in tables that are indexed by incident neutron energies [3].

2.6.3.1 Temperature Dependence

The G4NDL data used by the NeutronHP processes was evaluated at 0 kelvin. Therefore, for any realistic simulation, the NeutronHP physics processes use on-flight Doppler-broadening to determine the interaction cross sections at a given material temperature (see Section 2.3.2) [12]. This does slow down any simulation with materials above 0 K, with increasing inefficiency as the temperature of the material increases. Therefore, it is advantageous to create data libraries evaluated at higher temperatures if possible.

2.6.4 Data Processing

The data processing in Geant4 happens in four stages from the individual hits to the entire simulation (see Figure 1.1). The four stages of data collection and analysis are [3]

1. Hits are processed in a *sensitive detector* and saved in hit collections.
2. Hit collections for an event are gathered and processed in the *event action*.
3. Data from all events (processed by the event action) is further analyzed by the *run action*.
4. Data from the multiple runs may be collated and analyzed after the simulation has finished (*optional*).

To gain access to the data in a later stage, the data must be saved and passed to the subsequent stages. For example, the sensitive detector saves the pertinent information for individual hits in a hit collection, which is passed to the event action for further analysis.

2.6.4.1 Initial Actions: Hit and Event Level

Geant4 contains several scoring classes, but the most versatile is the sensitive detector class. In the definition of the world volumes, selected volumes are set as sensitive detectors. By doing so, any interaction (hit) that occurs in that volume is registered and may be analyzed/saved in the hits processing function of the sensitive detector. The pertinent information for each hit may then be saved in a hit object (container class), and added to the hit collection of the sensitive detector for the current event [3]. Much of the analysis, such as discriminating hits based on interaction type, occurs in this stage.

The event action allows the hit collections stored by each sensitive detector to be collated and analyzed. The data may be reduced (in size) and repacked in a new container class before transferring it to the run action to increase efficiency (especially important for event-level parallelism, Section B.2). Additionally, the event action may be used to complete any actions that need to be performed at the beginning or end at each event.

2.6.4.2 Final Actions: Run and Simulation Level

The run action is essentially the same as the event action, except that it is used to collect the data from the events, as well as completing any necessary actions at the

beginning or end of the run. In general, the run action is used to control the simulation in terms of initialization, output and data logging. Additional data processing may be done at the simulation level if the simulation includes multiple runs. However, this behaviour is not well supported by Geant4, so more user-generated code must be added for data analysis at this level.

Chapter 3

Related Research

As alluded to in Chapter 2, Geant4 exists in a larger ecosystem of simulation codes, including other Monte Carlo codes that are similar in many ways. Since Geant4 is a modern code that takes advantage of prior knowledge, such similarity is expected and even desired [3]. Furthermore, Geant4 has a diverse range of applications, although it is not widely used in reactor physics. One prominent example, however, is the use of Geant4 to model accelerator driven subcritical reactors (ADSR), where a particle accelerator provides the external neutron source necessary to sustain a stable neutron population in the reactor [21].

3.1 Related Simulation Codes

The following three simulation codes, MCNP, DRAGON and TART 2005 are used specifically in reactor physics. Each code has a significant history and has been well studied; moreover, they all overlap with the code developed for this thesis for certain applications. This is not a comprehensive list of nuclear codes; other codes, such as

WIMS and SCALE, are also widely used [22]. However, the following codes were used to validate the NStable code, with each code being used in a particular area where it was most applicable (see Chapter 5).

3.1.1 MCNP

MCNP (Monte Carlo N-Particle) is a three dimensional Monte Carlo neutron, electron and photon transport code that was developed by Los Alamos National Labs (LANL) [4]. MCNP5, the latest version in the MCNP series, was used in this thesis.

3.1.1.1 Application to Reactor Physics

MCNP is a general transport code, and thus, it can be applied to a large number of specific problems. These problems usually fall into two categories: physics simulations and criticality calculations. Fundamental physics is a broad category, covering most simulations where neutrons are fired into a test geometry, and the results of each shot are tallied for later analysis. For example, MCNP could be used in a detector simulation where the sensitivity of a particular design needs to be evaluated, especially if the detector produces photons or electrons as intermediaries. However, the limited number of particle types available does restrict the scenarios that MCNP can simulate (see Section 3.1.1.2 for the extended version of MCNP that seeks to resolve this issue).

MCNP is used extensively to validate deterministic codes through criticality calculations, as well as other characteristic properties, such as the neutron flux. As a Monte Carlo code, MCNP does not apply simplifications such as energy discretization (see Section 2.4.1), and therefore, is generally more accurate than deterministic codes (assuming the simulation is well posed). However, given the speed advantage of

deterministic codes, MCNP is not usually used for routine reactor calculations, such as determining a new fuelling scheme.

MCNP solves criticality problems using the generational method (*KCODE*) described in Section 2.5.6.1. This method is inaccurate outside of the near-critical regime, so MCNP should not be used to *accurately* calculate the criticality of sub-/supercritical systems. However, the generational method is sufficient for coarse qualitative calculations; MCNP will show that a system is sub- or supercritical, but the calculated value of k_{eff} will not be accurate [10]. Previous versions of MCNP did possess the ability to solve for the α-eigenvalue (*ACODE*), but this feature is not present in Version 5.

3.1.1.2 MCNPX

MCNP also features the MCNPX series, which separated from the main MCNP series at version 4B. It incorporates additional features not contained in the main series (the X in the name stands for eXtended). These features include elements seen in Geant4 such as exotic particle tracking (muons, neutrinos), and interchangeable physics models. MCNPX also features isotopic depletion calculations (burnup) using the CINDER90 module [23]. However, some concepts, such as charged particles moving in electric fields, are still under development [23]. The next version of MCNP plans to merge MCNP5 and MCNPX to combine the capabilities of both in a single code.

3.1.1.3 Application to the Project Objectives

It is conceivable to use MCNP to achieve the goals set out in this thesis. MCNP has the ability to stop neutrons at a given time in the simulation and save these neutrons to a source file [4]. However, any modifications of this source distribution would have to be done in post-processing between runs and outside of MCNP. Any post-processing that involves reading and writing files to the hard disks is prohibitively expensive compared to operations carried out in RAM (computer random access memory). Therefore, while MCNP may be able to accomplish the goals of this project, the solution would not be trivial and would likely be inefficient.

3.1.2 TART 2005

TART 2005 is a general purpose, three dimensional, coupled neutron-photon Monte Carlo transport code. It was created by Lawrence Livermore National Labs (LLNL), and builds on its predecessor, TARTND. The first version of TART was released in 1995 (TART95), and subsequent versions have been released at regular intervals until the latest version, TART 2005 [24]. While TART 2005 is relatively small in size compared to MCNP5 and Geant4, it is a complete program including the simulation code, a select set of nuclear data, and utility programs that perform auxiliary tasks such as checking the input geometry for errors or plotting the nuclear data.

Unlike MCNP5 and Geant4, TART 2005 does not use continuous energy nuclear data by default. Instead, it uses a 700 group multi-group data library with self-shielding, although continuous energy data is available should the user require it. Using a multi-group library in TART does reduce computation times, but the accuracy of such calculations will depend on the discretization of the energy spectrum and the

implementation of the self-shielding correction. The reduction in computation time using the multi-group library is approximately a factor of two compared to using the continuous energy nuclear data [24]. Therefore, the benefit of this simplification is not overwhelming.

TART 2005 bears similarity to the NStable code developed in this project; many of the fundamental concepts used in TART were used as a reference for the NStable code. TART can calculate k_{eff} using all three methods (see Section 2.5.6.1), including the dynamic method used in the NStable code. Like the NStable code, the dynamic method in TART 2005 uses periodic renormalization of the neutron population to keep the population within a manageable range. TART 2005 also features automatic updating of the run duration based on the average neutron removal time, so that a simulation is not wholly dependent on the initial user input. Therefore, TART 2005 is able to follow a population of neutrons in time as it evolves [10].

3.1.2.1 Application to Project Objectives

While TART 2005 is a general Monte Carlo neutron transport code with available source code, it lacks the flexibility and access that Geant4 grants its user. It also does not support the evolution of material or geometric properties with respect to time. The source code for TART 2005 is available, but it was not designed to be easily modified and/or overridden. Therefore, implementing new behaviour would require changes to the source code, which would limit the portability of the modified TART code. Additionally, the scheme of achieving the project objectives using MCNP (see Section 3.1.1.3) could be used with TART 2005, but the same disadvantages would apply.

3.1.3 DRAGON

DRAGON is a deterministic neutron transport code that solves the neutron transport equation for lattice cells in two- or three-dimensions. It was developed at École Polytechnique de Montréal to unify and improve upon several existing models and algorithms [18]. DRAGON was developed so that new methods and models could be easily added to the source code package. Thus, DRAGON allows the user to choose from several multigroup nuclear data libraries, and to choose the method of solution used in the tracking module (see the DRAGON user manual and physics manual for a detailed explanation of the solution methods employed by DRAGON) [18, 17].

DRAGON has the ability to [18]

- Calculate multigroup fluxes for each zone of the subdivided lattice cell in 2D and 3D
- Calculate homogenized cross sections for the entire lattice cell
- Calculate the criticality of a lattice cell using the k-eigenvalue method
- Calculate isotopic depletion (burnup) of fissionable isotopes in the lattice cell
- Employ resonance self-shielding corrections in the above calculations

However, given that DRAGON is a deterministic transport code, it must employ the simplifications given in Section 2.4.1, namely spatial, energy and angular discretization. Addtionally, DRAGON is a static code in terms of flux and criticality calculations, and a quasistatic code in terms of burnup calculations. Therefore, it would be difficult to model transient conditions in DRAGON, where the system is not stable and the higher order spatial modes are present. On the other hand, it is a

well validated lattice cell calculation code, which makes it a good comparison for the NStable lattice cell results for near-critical systems.

3.2 Related Time-Dependent Monte Carlo Simulations

Monte Carlo simulations are widely used in scientific research, and the ability to perform time-dependent simulations has not gone unnoticed. Other researchers have also been working toward time-dependent Monte Carlo neutron simulations, especially those involving burnup or transmutation calculations. Two examples of this work are discussed below: one developed a quasistatic Monte Carlo code for burnup calculations, and the other used Geant4 for modelling an Accelerator Driven Subcritical Reactor (ADSR).

3.3 Time-Dependent Monte Carlo

Shayesteh and Shahriari describe the development of a time-dependent Monte Carlo (TDMC) code that is able to calculate k_{eff}, the neutron lifetime, and the flux in time and space [25]. In the TDMC code, k_{eff} is calculated using methods similar to MCNP. In particular, k_{eff} is calculated with the generational method (time-independent) using absorption, collision and track length estimators for k_{eff} [4, 25]. Likewise, the neutron lifetime is calculated using an averaged tally of neutron lifetimes, and the flux is estimated by averaging the neutron track lengths in a volume divided by the volume itself [25]. However, time in the TDMC code is broken into generations (or "cycles").

Thus, the time-dependence is only accurate for stable (critical) systems. Once the system is no-longer near critical, the neutron lifetime will begin to vary significantly so that the criticality estimates will no longer be valid (see Section 2.2.3).

For near-critical systems, the TDMC code performs isotopic depletion (burnup) in time using the Bateman equations. At the end of each generation, the total number of neutrons, and the delayed neutron precursor concentrations, are updated using a method similar to the point kinetics approach [1, 25]. Shayesteh and Shahriari show that this method is equivalent to the point kinetics model for flux calculations of simple slab geometries with multiple regions. The TDMC code is also able to alter the properties of this slab by changing the thickness of various regions between generations, or by varying the fission cross section of the fuel with respect to time. In particular, these modifications were applied to a seven region slab of fuel surrounded by an absorber and then a reflector. This model showed good agreement with the point kinetics model in terms of relative flux calculations [25]. Additionally, Shayesteh and Shahriari report using two-group cross section libraries with six delayed neutron groups, although it is not reported whether the TDMC code can also use continuous energy cross section libraries. Thus, the TDMC code, as reported, is capable of time-dependent Monte Carlo simulations of near-critical systems.

3.4 Modelling of ADSRs in Geant4

Bungau, Barlow and Cywinski show that Geant4 can be used to model time-dependent behaviour in accelerator driven subcritical thorium reactors [21]. The simulated reactor features 215 rods of ThO_2 (thorium-232) fuel (approximately 14 tonnes) surrounded by a ZrO_2 reflector and lead shielding. The reactor itself is subcritical, and

requires a steady source of neutrons from an accelerator to reach a critical level. This neutron source is created by directing a 1 GeV beam of protons at a 60 cm lead target at the centre of the reactor so that neutrons are produced through spallation. This produces a flux of neutrons into the thorium-oxide fuel, which transmutes the thorium-232 into uranium-233 and causes fission in any existing uranium-233. Additionally, spontaneous fission within the reactor adds to the neutron economy. The authors show that both MCNPX and their Geant4 code predict that the reactor will be critical on spontaneous fission alone if the reactor fuel is 1.9% $^{233}UO_2$. They also predicted, using their Geant4 code, that the reactor requires 15% $^{233}UO_2$ to achieve criticality if spontaneous fission is ignored and only the spallation neutrons are used. This means that the spallation neutrons can never make the reactor critical; it would already be supercritical due to spontaneous fission if the $^{233}UO_2$ concentration was at 15% [21].

To simulate the ADSR in Geant4 and calculate its criticality, three filters were added to the sensitive detector: *G4SDTimeFilter*, *G4SDParticleWithTimeFilter* and *G4SDParticleWithVolumeFilter* [21]. This allows the sensitive detector to divide hit results in time and in space, allowing the neutron population to be counted for a given time period (bin). The exact algorithm used to calculate the criticality of the reactor is not reported, but criticality is defined as the number of neutrons in one generation relative to the number of neutrons in the past generation, implying an MCNP-like k-eigenvalue calculation [4, 21]. The entire neutron population is followed in time by virtue of the sensitive detector filters, although no population renormalization is used, so simulations that are not-near critical would suffer from unmanageable neutron populations over relatively short time periods; an example of the evolution

of a neutron population in the ADSR is only shown for 1.8 μs in a reactor configuration with $k_{eff} = 0.9927$ [21]. This simulation duration is short enough that the adverse affects of not renormalizing the neutron population will not be seen.

Chapter 4

Contribution and Methodology

Most of the work accomplished involved adding new classes to Geant4 to extend the functionality of the code. In particular, these new classes allow Geant4 to simulate time dependent neutron populations. The other main focus was to create nuclear data libraries at different temperatures by converting the MCNP data libraries into Geant4's G4NDL format using a Python script.

4.1 Code Development

The extensions for Geant4 developed in this project were written using a standard C++ IDE (Integrated Development Environment) called Code::Blocks. The code was then compiled externally using the standard g++ linux compiler and the Geant4 makefiles. Thus, the source code should compile and run on any standard Linux/Unix system; it has never been tested on a Windows system, so it may not work in that environment. The evolution of the NStable [1] code was tracked using the Bazaar

[1]The Geant4 simulation program created for this project was named "NeutronStability", and will be referred to as the *NStable code* in this thesis. A complete listing of the classes and files added

version control system. Bazaar provided a detailed history of the Geant4 extensions by documenting every major change, and it also provided an easy method for transferring the source code between various systems. The actual simulations were performed on an i-7 64-bit desktop PC, as well as a simulation cluster with two 32-processor computation nodes (64-bit AMD processors). The versions of the major software used in this project are listed below

Table 4.1: List of software used

Software	**Version**
Geant4	4.9.4.p02
MCNP	5
DRAGON	3.06J
TART	2005
g++	4.6.3-2
python	2.7.3
Bazaar	2.3.4

4.2 Geant4 Extensions

While Geant4 is a general purpose Monte Carlo code, it has seen little application in the field of nuclear engineering. Therefore, the Geant4 source code contains few tools to simulate the environments found in a nuclear reactor, although it does have many helpful utility classes that can be used to build such tools [12]. All of the extensions to the Geant4 source code that are described in this section did not exist *in Geant4*, and were created specifically for this project.

to Geant4 to make the NStable code is given in Appendix C. The appendix also includes a short description for each class or file.

As outlined in Section 1.1, the objective was to simulate the evolution of a set of neutrons and their descendants over a period of time on the order of seconds. During this simulation, the material and geometric composition of the simulation world must also be able to change. This behaviour is different from traditional Monte Carlo detector simulations where single particles are fired into the simulation world, creating a cascade of events, and then the end result is recorded and analyzed [3]. Instead, this project seeks to analyse the system at regular intervals without interrupting the evolution of the neutron population. Therefore, the necessary extensions to achieve this goal are not trivial.

4.2.1 Neutron Population Stabilization

Nuclear reactors are by necessity multiplying mediums, where the neutron population will either grow or shrink exponentially given the shape of the reactor and the ratio of neutron absorbers to neutron sources (fissionable isotopes) [1]. If the simulation world, is sub- or supercritical, then the neutron population will continue to shrink or grow exponentially according to

$$N(t) = N_0 \ \exp\left[\left(\frac{k_{eff} - 1}{l}\right) t\right] \tag{4.1}$$

where $N(t)$ is the neutron population with respect to time, and l is the average neutron lifetime [1]. Note that Equation 4.1 assumes that l and k_{eff} do not change with time, which is true as long as neither the material or geometric composition of the simulation world changes. Using this equation, the population will change by a factor of $\exp(k_{eff} - 1)$ every l seconds, and since l is on the order of nanoseconds to milliseconds, the neutron population can change significantly in a short time. Such

a population will either grow to exceed the available computer memory, or vanish completely. With shrinking populations, the time until the population vanishes can be increased by increasing the number of initial particles, but this quantity is also limited by available computer memory. Moreover, once the population becomes small enough, the average spatial distribution of neutrons in the system will no longer be adequately defined by the current neutron population. Therefore, the *simulated* spatial distribution will change as the remaining neutrons move about the system, causing fluctuations in k_{eff} that *are not physical*; no material or geometric changes occur, so the fluctuations in k_{eff} only occur because the initial neutron population was too small. Consequently, any simulation of a neutron population in time must contain some method to stabilize the population within an appropriate range.

4.2.1.1 General Approach

The flowchart in Figure 4.1 shows the algorithm used to stabilize the neutron population. Each step will be explained in more detail in the following sections. Basically, the simulation is broken into runs of length T_{run}, and at the end of each run, the population is renormalized to the initial neutron population, N. As long as the neutron population does not change too much over the run (i.e. $N' >> N$ or $N' << N$), then this approach will keep the neutron population within a manageable range. Since the neutrons at the end of one run become the primaries at the beginning of the next run, the evolution of the neutron population is continuous with respect to time (see Section 4.2.1.5 for a discussion of the effect of the renormalization on the continuity of this evolution).

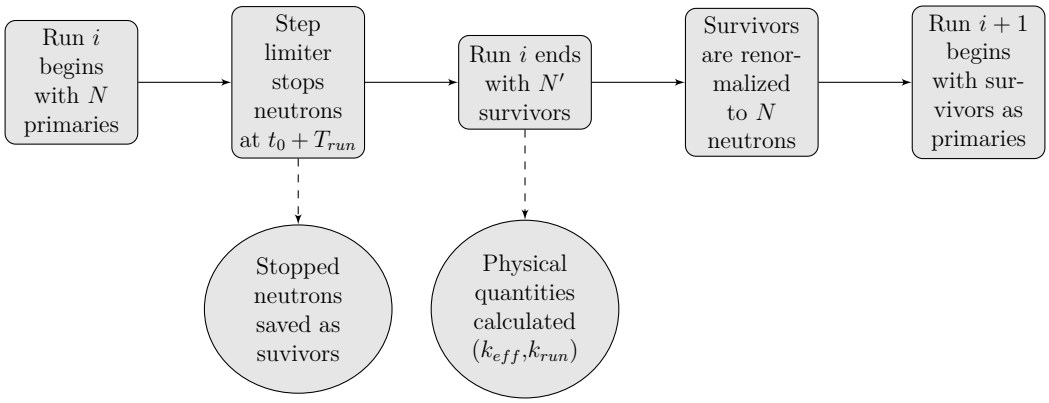


Figure 4.1: High level simulation flow diagram for Geant4

This general strategy, and its difference from the approach used in other Monte Carlo codes such as MCNP, is illuminated in Figure 4.2. In the NStable code, the simulation is comprised of many sequential runs, which serve as the discrete time intervals necessary for the renormalization. All of the neutrons are stopped at the same time at the end of the run, so the time-dependence of the simulation is not affected. In contrast, k-eigenvalue calculations in Monte Carlo codes are usually performed by comparing the change in fission neutrons per generation [4]. Therefore, a single run, or cycle, in such a code (e.g. MCNP) uses the neutron generations to divide the simulation, which is shown by the dotted line on Figure 4.2. While the average neutron generation time of the system is usually constant (average lifetime of neutrons causing fission), the individual lifetimes of the neutrons vary. Therefore, the current cycle ends at a different time for each neutron. This convention is reasonable as long as the system is time-independent (critical), but for time dependent systems, this approach will not produce accurate results (see Section 2.2.3).

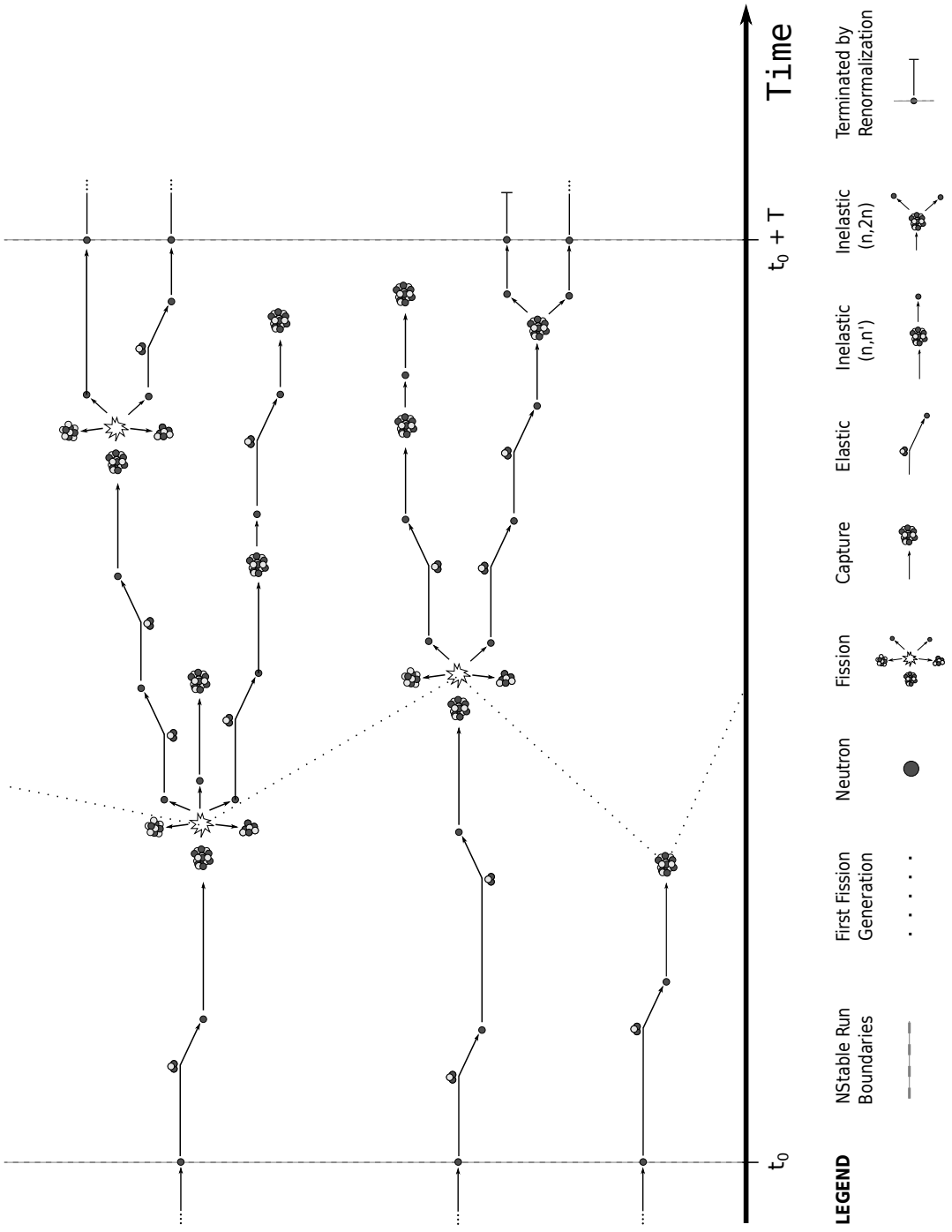


Figure 4.2: Division of the simulation in time.

4.2.1.2 Division of the Simulation into Discrete Intervals

To stabilize the neutron population, the NStable code divides the simulation into discrete time intervals and renormalizes the neutron population at the end of each interval, where

$$N(t_0 + T_{Run}) \xrightarrow{norm} N(t_0) \tag{4.2}$$

after renormalization and t_0 is the time at the beginning of a run. However, the default Geant4 run manager (*G4RunManager*) only instantiates a single independent run. Therefore, to implement sequential, dependent runs, the default run manager was replaced by a new *NSRunManager* class that was derived from *G4RunManager*. This new run manager simulates a given number of sequential runs, and handles the communication between the current run and the next one. To quantify the population change over a single run, the *run multiplication constant*, k_{run}, is defined as

$$k_{run} \equiv \frac{N(t_0 + T_{run})}{N(t_0)} \tag{4.3}$$

This is simply a measure of the overall neutron population change during the current run, where the population was

$$k_{run} = \begin{cases} > 1 & \text{growing} \\ 1 & \text{stable} \\ < 1 & \text{shrinking} \end{cases}$$

depending on k_{run}.

4.2.1.3 Stopping the Neutrons at the End of the Run

By default in Geant4, neutrons do not stop with respect to time; they stop with respect to step length proposed by the winning post-step process. Therefore, to get a neutron to stop at a prescribed time, such as the end of the current run ($t_f = t_0 + T_{run}$), a new step limiter process (a class derived from *G4StepLimiter*) was added in the NStable code. This process, *NSStepLimiter*, always proposes a step length equal to the distance the neutron can travel before the run ends. That is

$$d_{limit} = (t_f - t_{step})\ v \tag{4.4}$$

where t_{step} is the time at the beginning of the current step, and v is the scalar velocity of the neutron. Unlike other step limiting processes, the *NSStepLimiter* process is not truly a null interaction (see the next Section 4.2.1.4); it kills the neutron to remove it from the simulation. Therefore, no neutrons will survive past the end of the run because the *NSStepLimiter* process is *always* the winning process if the neutron reaches t_f.

The duration of a run should be chosen such that the neutron population does not undergo significant change that would degrade the performance of the simulation. As a rule of thumb, the run duration should be chosen given the following condition

$$\left\{ T_{run} \ \mid \ N(t_f) \in [\, 0.5N(t_0),\, 2N(t_0) \,] \right\} \tag{4.5}$$

This range for T_{run} is arbitrary, but has proved sufficient during the validation testing of the NStable code.

4.2.1.4 Survivors

Those neutrons that reach the end of a run and are stopped by the *NSStepLimiter* process are deemed to be the survivors of the run. These survivors then become primaries in the next run to create a continuous evolution of the neutron population. To completely replicate the survivors in the next run, the following information must be recorded

- Position
- Momentum
- Current time
- Lifetime
- All η_λ^i values

While the *NSStepLimiter* process is not a null interaction because it kills the neutrons it acts upon, it is an arbitrary division of the current step (over two runs). Thus, it should not affect the physics of the simulation, so each of the η_λ^i values must be decremented, recorded and used when the neutron begins tracking in the next run. For easy transfer of this data, especially between the master (computer) process and its slaves in a parallel computation, these values are saved to the *NeutronData* container class. Once saved, the *NeutronData* object represents a single survivor.

4.2.1.5 Renormalization

To renormalize the neutron population to the number of initial primary neutrons, selected survivors are either duplicated or deleted depending on the value of k_{run}. These selected neutrons are referred to as the *targets* of the duplication and deletion algorithm. The selection algorithm is given below [26]

Algorithm 4.1: Target selection for deletion or duplication

Input: Number of primary neutrons N_p, number of survivors N_s, list of survivors

Output: List of targets for deletion or duplication

$m = |N_s - N_p|$ `// Number of missing/extra neutrons`

for $i = 0 \rightarrow (N_s - 1)$ **do**

  $r \in [0, 1]$ `// Generate a random number between 0 and 1`

  if $r < \frac{m}{N_s - i}$ **then**

    Add survivor i to list of targets

    $m = m - 1$

  end

end

This selection algorithm is uniformly applied across the survivors without bias, so the renormalization process does not change any properties of the neutron population such as the spatial distribution or the energy spectrum (see Appendix D for a more complete discussion of the target selection algorithm). Therefore, the continuity of the population with respect to time is affected by the renormalization; since the NStable code uses a Monte Carlo method, the individual neutrons represented a sample of neutrons that could exist in the real system, and therefore, are not *individually* important so long as the sample size is large enough to represent the macroscopic system properties.

The renormalization algorithm occurs in the primary generator action (*NSPrimaryGenerator*) at the beginning of each run. The survivors are stored as a doubly-linked

list of *NeutronData* objects. Thus, deletion or duplication of the targeted survivors only requires that the algorithm remove survivors from the list or add copies of survivors to the end of the list (see Appendix A for details on doubly-linked lists). Then the renormalized survivors are used as templates for the primaries produced by the *NSPrimaryGenerator* in the next run.

4.2.1.6 Initializing the Next Run

Once the list of survivors has been renormalized to the number of initial primaries, these survivors can be used directly to produce the primaries for the next run. The *NeutronData* for each survivor is unpacked and used to set the position, momentum, lifetime, and current time for a primary neutron. Since the η_λ^i values are stored by the processes, and not by the individual neutrons, these values must be stored in each primary neutron using the *G4VPrimaryParticleInformation* pointer provided by Geant4. This pointer is a data member of the primary particle and allows a container class to be attached to it that can store the η_λ^i values. The container class, *PrimaryNeutronInfo*, was derived from *G4VPrimaryParticleInformation* for this purpose. After all the survivors have been used to create primary neutrons, the simulation of the current run begins.

In general, the η_λ^i values are initialized for all applicable processes using Equation 2.34 when the simulation begins tracking a new neutron. Therefore, new processes had to be derived from the NeutronHP processes to add the capability of initializing the η_λ^i values to the values stored in the *PrimaryNeutronInfo*. When these new processes (e.g. *NSHadronElasticProcess*) start tracking a new particle, they check to see if it is a primary neutron. If so, the process extracts its specific η_λ^i value from the

PrimaryNeutronInfo object and uses this as the initial number of mean path lengths left.

4.2.1.7 Delayed Neutrons

Delayed neutrons are not generally produced in the current run because they are born after the fission that created them (possibly seconds later). Therefore, unless the delayed neutrons are being produced instantaneously, they must be saved and added to future runs. The delayed neutrons are recorded as *NeutronData* objects in a separate delayed neutron list. The "current time" of a delayed neutron is not the time of the fission that created it, but rather the time when it is born, after the fission. At the start of each run, the primary generator scans the delayed neutron list to see if any delayed neutrons should be born in the upcoming run. If so, these delayed neutrons are removed from the delayed neutron list and added to the survivors list *prior* to renormalization. The delayed neutrons need to be renormalized because they were produced during a run when the neutron population had deviated from its normalized value. Therefore, the fraction of delayed neutrons to prompt neutrons reflects the instantaneous population, not the normalized value. If the delayed neutrons survive the renormalization process, then they will be treated like any other primary in the simulation.

Given that the delayed neutrons are born long after the fissions that created them, achieving a stable distribution of delayed neutrons would require simulating the neutron population for a total simulation time greater than ten seconds. This is a relatively long duration when the run durations are on the order of nanoseconds to milliseconds. Consequently, this capability has not been fully tested; instead, the

instantaneous production of delayed neutrons has been used as an approximation. This approximation is sufficient for criticality calculations, especially calculations for systems that are near-critical ($k_{eff} \approx 1$), but it can be undesirable for transient simulations.

4.2.2 Analysis

The analysis of the simulation data in the NStable code is performed at each of the levels listed in Section 2.6.4. Each level of analysis required a new class to be derived from basic Geant4 classes for sensitive detectors, event and run actions, and run managers.

4.2.2.1 Neutron Sensitive Detector

A single general sensitive detector called *NeutronSD* is applied to every volume in a NStable simulation. This sensitive detector is used to kill any particles other than neutrons (not useful for the current objectives[2]), record the survivors and delayed neutrons, and tally important physical quantities for the simulation. These quantities (over one event) are

- Number of neutrons produced
- Total lifetime of lost neutrons
- Number of neutrons lost
- Position of each fission

The total lifetime is the sum of all the lifetimes of the neutrons when they were killed. It is tallied so that it may be averaged over *all* the neutrons lost during

[2]Photons produced from neutron-nuclear interactions will in turn produce a group of delayed neutrons known as photo-neutrons. However, the fraction of photo-neutrons is relatively small compared to conventional neutrons, although they can be important in heavy water moderated systems. For simplicity, all photon interactions were ignored.

the run. Delayed neutrons that do not occur in the current run are not counted as production until the neutron is actually born in a future run. Additionally, (n,*n) inelastic collisions are only counted for the number of neutrons produced beyond the incident neutron (see Section 2.1.1). Technically, the incident neutron is absorbed momentarily, but this is not a true loss and production of neutrons; the neutron economy is unchanged by the absorption and emission of the incident neutron. All of this data is saved in a *TallyHit* class, which is saved to the hit collection for the current event (see Section 2.6.4).

4.2.2.2 Event Action

The event action in NStable exists to repackage the *TallyHit* data for the event into an *EventData* object. This repackaging is more important for parallel processing (see Section 4.2.5), but it is advantageous to save the data in the event action where it will persist until the next event begins. It is incumbent on the run manager to send this data to the run action before it is overwritten. An *EventData* object contains the following information from the current event

- Number of neutrons produced
- Number of neutrons lost
- Total lifetime of lost neutrons
- Position of each fission
- List of survivors
- List of delayed neutrons
- Event identifier

where the event identifier is a unique integer designation given to each event by the run manager.

4.2.2.3 Run Action

The run action is responsible for the data collection, analysis and output that must be completed at the end of each run. First, the run manager passes the results of each event (*NSEventData*) to the run action. The tallies are simply added to running totals, but the survivors and delayed neutrons for each event are kept separated. Once all the events have been completed, the survivors from each event are collated into a single list that is ordered by the event identifiers; the same procedure is used to order the delayed neutrons. This ensures that the survivor and delayed neutrons are consistently ordered regardless of the order in which the event data was passed to the run action. Again, this is important for parallel processing.

After the data from each event has been recorded and organized, the run action calculates the following

1. The run multiplication constant, k_{run}
2. The Shannon entropy, H_{src}
3. The average neutron lifetime, l
4. The neutron multiplication constant, k_{eff}
5. The run duration

Where the run duration is measured using a *G4Timer* object, which is started at the beginning of the run and stopped once the above calculations are finished. Finally, all of this data, and other key quantities such as the number of neutrons produced, is written either to a log file or to the screen.

4.2.2.4 Run Manager

Since runs in Geant4 are independent, communication between successive runs must be handled either by the run action or the run manager. Since the run manager also has access to the other user actions (e.g. event action, primary generator action), it is advantageous to have the run manager facilitate most of the communication between the user action classes. Additionally, the default run manager only allows one run per simulation, and therefore, it was not sufficient for the NStable code. For these reasons, the *NSRunManager* class was derived from the standard *G4RunManager* class.

At its most fundamental level, the *NSRunManager* class is a loop over N runs. Once these runs finish, the simulation performs any final actions and then stops. However, within this loop, the run manager does the following

1. **Before each event:**
    - Gets the primaries for each event from the primary generator action
2. **After each event:**
    - Passes the event data from the event action to the run action
3. **After the run:**
    - Updates the simulation time (e.g. start and end of next run)
    - Passes the survivors and delayed neutrons to the primary generator action
    - Checks for convergence in the Shannon entropy (Equation 4.6)
    - If converged, keeps a running tally of physical quantities (e.g. k_{eff})

- Saves the survivor and delayed neutron lists to a text file (only at specified intervals)

After all the runs have been simulated, the run manager averages the important quantities over the number of runs since convergence was achieved. These averages are either printed to the screen or to the log file.

Convergence in the Shannon entropy, and by extension, the spatial distribution of neutrons, is achieved if the Shannon entropy from the past N_C runs is within a given convergence limit, L_C. That is

$$L_C > \left| \frac{H_i - H_j}{H_i} \right| \qquad \forall j \in [i - N_c, i - 1] \tag{4.6}$$

where H_i is the Shannon entropy of current run (run i). The default values for N_C and L_C are 25 and 1.0% respectively (see Section 5.5.1 for examples of Shannon entropy convergence).

4.2.2.5 Main Driver File

The main driver file (the file containing the *main* function) is the foundation of the entire NStable code (see Appendix F.1.1 for the complete driver file). The driver file, *NStable.cc*, does the following

1. Reads the input variables from the input file (see Section 4.2.6)
2. Instantiates the run manager, which instantiates the Geant4 kernel
3. Instantiates the main user classes: the simulation world, the physics list, and the primary generator

4. Instantiates the optional user actions: the event action, the run action
5. Sets the output stream (either to the screen or to a log file)
6. Starts the simulation with the run manager's *BeamOn* function
7. Performs any remaining tasks after the simulation ends (clean up)

The output stream is either set to the standard Geant4 output, *G4cout*, which prints to the screen, or to a log file created by the *LoggingAction* class. The *LoggingAction* class is designed to not only create (or open) the given text file, but also to provide an output stream to the log file so other classes can print to it as well.

4.2.3 Simulation Worlds

The core NStable code does not depend on the composition of the simulation world. It is flexible enough to handle a diverse range of systems including a solid sphere of a given material and a reactor lattice cell in an infinite lattice. The simulation world in the NStable code is built using the *NSWorld* class, which manages constructors for each available world type. Based on the world type that the user chooses, the *NSWorld* class calls the appropriate world constructor.

All of the simulation worlds are built from the *NSWorldConstructor* base class. This base class contains all common functionality between the world classes and it also allows the *NSWorld* class to use polymorphism to simplify its interactions with the chosen world constructor (i.e. it does not need a pointer to each world constructor type). The world constructors currently available are:

Bare Sphere

The *BareSphereConstructor* creates a sphere of a homogeneous material

with a given radius and material temperature. Available materials include pure uranium-235 (U235), natural uranium (NU), and a mixture of heavy water (D_2O) and natural uranium, referred to as the UHW mixture. In the case of the mixtures (NU and UHW), the relative isotopic concentrations are set. However, the radius can be adjusted to change the criticality of the system.

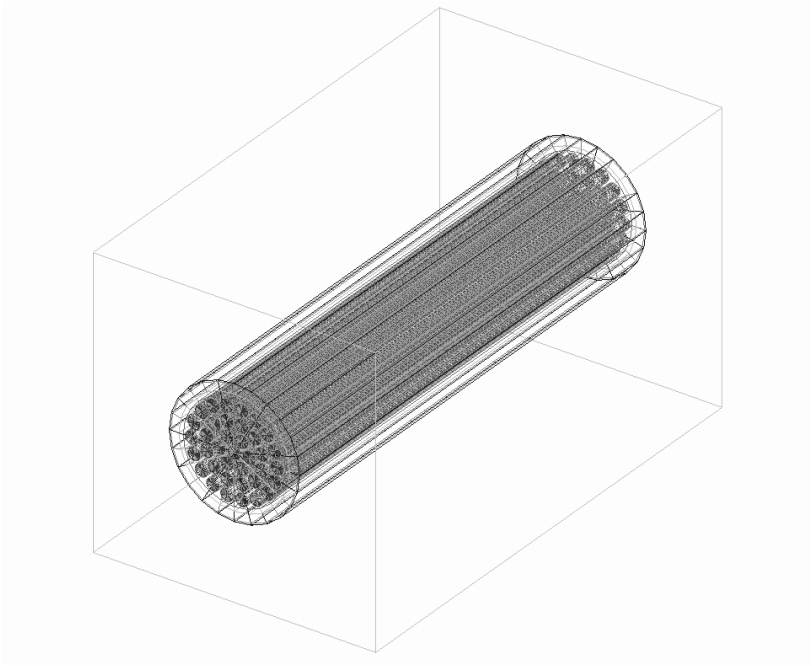
Homogeneous Lattice Cell

The *CubeConstructor* is the infinite analog of the *BareSphereConstructor*. Using periodic boundary conditions (see Section 4.2.4.1), the cube constructor creates an infinite lattice of homogeneous cubes. Available materials include NU and UHW, where the relative isotopic concentrations can be adjusted for varying degrees of criticality.

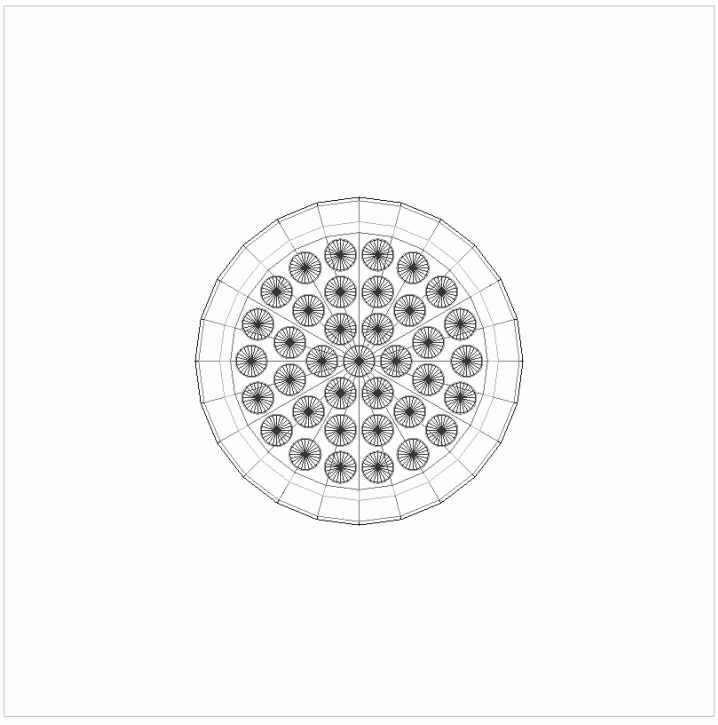
CANDU 6 Lattice Cell

The *C6LatticeConstructor* is based on an idealized CANDU 6 lattice cell with fresh fuel and no instrumentation or adjusters (see Figure 4.3) [18]. Like the homogeneous lattice cell, the CANDU 6 lattice cell uses periodic boundary conditions to simulate an infinite lattice. The lattice pitch, fuel temperature, coolant temperature, and coolant density can all be adjusted for this constructor.

These simulation worlds have already been built and are included in the NStable code, but any end-users may define their own simulation worlds by creating new derived classes from the *NSWorldConstructor* base class.



(a) Side View



(b) Front View

Figure 4.3: CANDU 6 lattice cell geometry in Geant4.

4.2.3.1 Dynamic Simulation Worlds

Most of the adjustments to the simulation world listed above occur during the construction of the simulation world and remain constant for the actual simulation. However, some properties may be changed incrementally over the course of the simulation to approximate a continuous change in these properties. This allows the NStable code to model transient changes in material and geometric properties so that the resulting feedback can be studied (e.g. k_{eff} change in response to fuel temperature). The procedure for these incremental changes is

Algorithm 4.2: Incrementally change a geometric or material property

Input: Distribution for property as a function of time, $p(t)$
Output: Updated property value, p'

```
/* Current run ends                                          */
t = t_f                              // Update time to end of run
p' = interp(p(t), t_f)          // Interpolate p(t) to find p' at t_f
GeometryHasChanged()              // Inform run manager of change
/* Next run begins                                           */
```

The distributions for the property being changed are set in an input file as a two dimensional table of data (see Section 4.2.6). The third step in Algorithm 4.2, informing the run manager of a change in a geometric or material property, is required so that the run manager can recalculate tables of combined material properties (e.g. total isotopic density) that the tracking algorithms use to speed up calculations. Currently, the incremental changes are limited to temperature and density changes of select materials. The materials that allow incremental temperature changes are

- Homogeneous material (Bare Sphere and Homogeneous Lattice Cell)
- Fuel (CANDU 6 Lattice)
- Coolant (CANDU 6 Lattice)

Incremental density changes are limited to the coolant of the CANDU 6 lattice. Again, these adjustable properties simply represent the current dynamic capabilities of the NStable code. End-users of the code can add additional dynamic properties by defining their own simulation world and modifying the *ParseInput* class (Section 4.2.6).

4.2.4 Physics Lists

The *HPNeutronPhysicsList* class, derived from *G4VUserPhysicsList*, is responsible for declaring and initializing the physics models used in the simulation. All neutron interactions with energies up to 20 MeV in the NStable code are modelled with the NeutronHP processes. As mentioned in Section 4.2.1.6, new physics processes were derived from the NeutronHP processes to allow the η_λ^i values to be set at the beginning of tracking. For energies above 20 MeV, a low energy parameterised model (*G4LEPNeutronBuilder*) is used for all non-elastic collisions up to 25 GeV, except for inelastic collisions from 20 MeV to 9.5 GeV, which use a Bertini intranuclear cascade model (*G4BertiniNeutronBuilder*). For elastic collisions between 20 MeV and 20 GeV, a simple elastic model, *G4LElastic* is used. While neutrons in a reactor rarely reach energies above 20 MeV, these models need to be defined so that neutrons in that energy range can be simulated if the need arises. Additionally, the physics list is also used to instantiate the *NSStepLimiter* and *BoundaryStepLimiter* processes (see Sections 4.2.1.3 and 4.2.4.1).

For the NeutronHP processes, the nuclear cross section data is parsed and sampled with the *NeutronHPCSData* class. This class is derived from the *G4VCrossSectionDataSet* class, and is based on the default cross section data classes used with the NeutronHP processes (e.g. *G4NeutronHPElasticData*). Four default data classes are defined (e.g. elastic, inelastic, fission and capture) even though most of the code within each class is identical. Moreover, the default data classes assume the cross sections were processed at 0 kelvin. This is true for the G4NDL data libraries, but for a data library processed at a higher temperature, the on-flight Doppler broadening produces the wrong result. If the default data class received cross sections for data processed at 300 kelvin, and the current material temperature was also 300 kelvin, then the default Doppler broadening algorithm would still try to broaden the data by 300 degrees, essentially finding the cross section at 600 kelvin. Therefore, the new *NeutronHPCSData* class was created to not only handle cross section data sets for any neutron interaction type, but also to Doppler broaden the cross sections based on a given evaluation temperature.

4.2.4.1 Periodic Boundary Conditions

Periodic boundary conditions were implemented in the NStable code to produce infinite lattices of the world volume. With a periodic boundary, a neutron that leaves the world will reappear on the opposite side of the world with the same momentum; this approximates the neutron entering a second identical lattice cell. If the centre of the world volume is at the origin, this is formalized as

$$\overrightarrow{x} \rightarrow f\Big(\hat{n}(\overrightarrow{x})\Big) \cdot \overrightarrow{x} \tag{4.7}$$

where $\overrightarrow{x}$ is a position on the edge of the world volume, $\hat{n}(\overrightarrow{x})$ is the outward normal to the surface of the lattice cell at $\overrightarrow{x}$, and $f(\hat{n})$ is given by

$$f(\hat{n}) = \Big(f(i), f(j), f(k)\Big) \quad | \quad f(i), f(j), f(k) = \begin{cases} -1 & \text{if} \quad i, j, k \neq 0 \\ 1 & \text{otherwise} \end{cases} \tag{4.8}$$

where i, j, and k are the components of $\hat{n}$ in the cardinal Cartesian directions. Equation 4.8 essentially changes the sign of any component of $\overrightarrow{x}$ that is parallel to $\hat{n}$. For example, if the neutron left the world at $\overrightarrow{x} = (5, 1, -2)$ through a surface normal to $\hat{n} = (1, 0, 0)$, then Equation 4.8 states that

$$\overrightarrow{x} = (-1, 1, 1) \cdot (5, 1, -2) = (-5, 1, -2) \tag{4.9}$$

This assumes that the world is symmetric about the origin; otherwise, Equation 4.7 would need to be more complex.

The boundary condition is implemented using the *BoundaryStepLimiter* process. For any neutron leaving the world volume, the *BoundaryStepLimiter* proposes the minimum step size possible ($10^{-32}mm$). When it acts on a neutron, it produces a secondary neutron on the opposite side of the world volume that is an exact copy of the original neutron, and then the original is killed. It is necessary to produce a secondary neutron because "teleporting" the original neutron across the world volume would cause errors in the tracking algorithms.

4.2.5 Parallel Implementation

The implementation of parallel processing in the NStable code uses TOP-C and Marshalgen as described in Section B.2. The conversion of the NStable code to a parallel version required two steps. First, a new run manager (*ParNSRunManager*) was created to incorporate communication between the master and the slaves. Second, Marshalgen was used to create marshallable versions of the *NeutronData*, *NSEventData* and *NSPrimaryData* container classes, which are used to transfer data between the master and the slaves. A new driver file, *NStable.icc*, was also created to initialize and finalize TOP-C. It takes three steps

1. Initialize TOP-C
2. Call the *main* function from NStable.cc
3. Finalize TOP-C

The new driver file also overwrites every instance of *NSRunManager* with *ParNSRunManager*. Note that these overwrites take place *after* the *ParNSRunManager* class has been declared because *ParNSRunManager* inherits from *NSRunManager*.

4.2.5.1 Parallel Run Manager

All parallel processes in TOP-C have three main functions centred around generating, performing and checking the results of tasks (see Section B.2). The implementation of these three functions in the NStable code is described below

Generate Task Input → *GenerateEventInput()*

The master generates an *NSPrimaryData* object with a random seed, an event designation (integer number), and a vector of primary neutrons. The

primary data also includes any incremental changes in material or geometric properties of the simulation world. Then the *NSPrimaryData* object is marshalled and sent to a slave. The random seed is generated by the master so that the event will always achieve the same result regardless of the slave it is processed on.

Do Task → *DoEvent()*

The slave unmarshals the *NSPrimaryData* object and sets all primary data for the event. This includes the primary neutrons and the random number generator seed, as well as any changes to the simulation world. The event is then processed, and the resulting *NSEventData* object is marshalled and sent back to the master.

Check Task Result → *CheckEventResult()*

The master unmarshals the *NSEventData* object and passes it to the run action for processing.

In this implementation, the master controls every aspect of the simulation, including the random number generator and simulation world of the slaves. This centralized command is advantageous because the master always knows what state the slaves are in, and the slaves are only responsible for processing single events. For this reason, the slaves do not have a run action, nor do they perform any renormalization of the neutron population. All of the data necessary to process the events, other than the base simulation world and the physics list, is provided by the master. Note that all of the slaves are initialized with the same input file as the master.

4.2.6 Simulation Parameters and Options

The NStable code uses a variety of parameters and options that are set by the user in the input file (see Appendix F.1.2 for an example input file). The input file arguments fall into three categories: required parameters that must be defined in every simulation, world specific parameters that revert to default values if not explicitly specified, and simulation options that enhance the user's control over the simulation. Additionally, a few arguments are set in the Linux environment variables; these arguments pertain either to the Geant4 source code or the makefiles.

The *ParseInput* class is responsible for parsing the input file at the start of the *main* function before any of the Geant4 classes are instantiated (e.g. the run manager, the simulation world, etc). It also acts as a container class by storing all of the input arguments. Rather than passing several arguments to each Geant4 class instantiated in *NStable.cc*, the classes are given a pointer to the *ParseInput* object. This allows each class to use any of the input arguments without having to specify which ones it needs in *NStable.cc*. A list of the available input parameters is given in Appendix E.

4.3 Geant4 Source Code Modifications

Generally, modifications of the Geant4 source code should be avoided if at all possible; any modifications damage the interoperability of the code on different computers. However, two instances were found during this project where a source code modification was the only reasonable solution. In the first case, a parameter was hard coded into the Geant4 source code that reduced the accuracy of every simulation. In the second case, the source code contained a bug that caused the code to crash. For

NStable to function properly, these source code modifications must be made to each Geant4 installation that will be used to run the NStable code.

4.3.1 Arbitrary Data Thinning

When data is stored in a *G4NeutronHPVector* object, the data is thinned using Algorithm 4.3. The algorithm tests whether the interpolation between two points (x_1, y_1) and (x_2, y_2) is accurate enough to ignore any intervening points $x_k \in (x_1, x_2)$. The accuracy of the interpolation must be with a precision limit p, which is set to 2% [12]. This is large enough that the data thinning causes noticeable differences in the results of simple simulations (see the example below). If it were possible to change the data thinning precision, a source code modification would be unnecessary, but there is currently no way of doing this. The source code contains a note stating that the optimization of the precision limit is ongoing, but no changes have been made in the latest revision of the code (version 4.9.5.p01) [13].

The effect of the thinning algorithm was tested using a simple Monte Carlo simulation in Geant4 and MCNP5. A beam of thermal neutrons was shot at a small cube of uranium-235 that measured 1 cm on each side, and the kinetic energy of any secondary neutrons leaving the cube was recorded. Thus, the energy spectrum of the secondary neutrons could be recreated for any given data library in either MCNP or Geant4. The MCNP results were used as the benchmark, and they were produced using the MCNP endf66 library (ENDF/B-VI cross sections at 293.6 K). These results were compared to Geant4 results produced from the G4NDL 3.14 and 4.0 data libraries with the thinning algorithm either enabled or disabled (commented out in the source code). This comparison is shown in Figure 4.4.

Algorithm 4.3: Data thinning in the *G4NeutronHPVector* class

Input: Vector of data of length N ($data$), last data point j saved to thinned buffer, precision p

Output: Thinned vector of data

```
for i < N do
    (x_1, y_1) = data(j)
    (x_2, y_2) = data(i)
    /* Check all points between j and i                    */
    for k ∈ (j, i) do
        (x_k, y_k) = data(k)                  // Current data point
        y = interp(x_1, x_2, y_1, y_2, x_k)    // Interpolate at x_k
        if |y − y_k| > |p × y| then
            Save data(k) to buffer
            j = k                          // Set last saved point
        end
    end
    i++                              // Move to next point in vector
end
data = buffer                          // Overwrite data with buffer
```

With the data thinned to a precision of 2% using Algorithm 4.3, the energy spectrum of the secondary neutrons deviated significantly from the MCNP spectrum regardless of the version of the G4NDL library. Disabling the thinning algorithm did not change the results when using version 3.14 of the G4NDL library, which implies that this version of library has already been pre-thinned to a precision of 2%. Conversely, disabling the thinning caused the G4NDL 4.0 results to match the MCNP spectrum. Therefore, the G4NDL 4.0 data library will provide more accurate results than the version 3.14, assuming that the data thinning is disabled in the *G4NeutronHPVector* class.

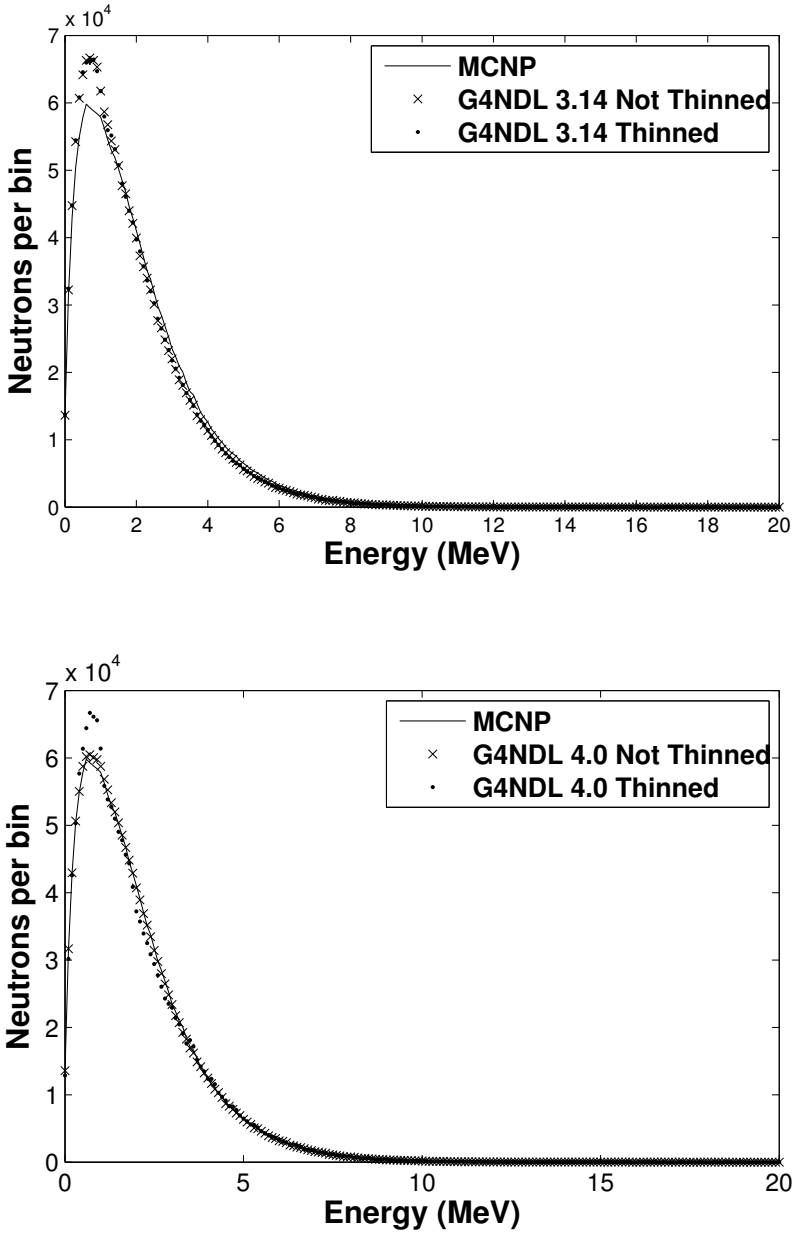


Figure 4.4: The effects of data thinning on the energy spectrum of secondary neutrons from neutron-U235 interactions using the G4NDL 3.14 (top) and 4.0 (bottom) data libraries as compared to MCNP.

4.3.2 Bug in Energy-Angular Definition

The following code from the *Sample()* function of *G4NeutronHPContAngularPar* class causes a runtime error [12].

Listing 4.1. Source code from *G4NeutronHPContAngularPar::Sample()*

```
474  for(i=0; i<nAngularParameters; i++)
475  {
476      theBuff1.SetX(i, theAngular[it−1].GetValue(i));
477      theBuff1.SetY(i, theAngular[it−1].GetValue(i+1));
478      theBuff2.SetX(i, theAngular[it].GetValue(i));
479      theBuff2.SetY(i, theAngular[it].GetValue(i+1));
480      i++;
481  }
```

The first problem is that i increments by two for every iteration of the loop. Therefore, the *SetX()* and *SetY()* functions try to set elements $[0, 2, 4, ...]$ of the buffer objects (*theBuff1* and *theBuff2*). This causes a runtime error since elements are skipped in these buffers. Additionally, the angular data, *theAngular*, has the following format

$$theAngular = \{P_{eng}, \cos_1\theta, P_1, \cos_2\theta, P_2, ...\}$$

where P_{eng} is the probability of the outgoing neutron energy (which is stored elsewhere), $\cos_i\theta$ is the i^{th} outgoing angle, and P_i is the probability of the i^{th} outgoing angle [12]. Therefore, the $GetValue()$ function should start at 1, not 0.

Fixing this bug is relatively simple, and the solution used in this project is shown in Listing 4.2.

Listing 4.2. Fixed source code from *G4NeutronHPContAngularPar::Sample()*

```
474  for(i=0, j=1; i<nAngularParameters; i++, j+=2)
475  {
476      theBuff1.SetX(i, theAngular[it−1].GetValue(j));
477      theBuff1.SetY(i, theAngular[it−1].GetValue(j+1));
478      theBuff2.SetX(i, theAngular[it].GetValue(j));
479      theBuff2.SetY(i, theAngular[it].GetValue(j+1));
480  }
```

In the fixed code, a separate index j is used to get the $cos\theta$ data from *theAngular*. Again, this bug is present in the latest version of Geant4 [13].

4.4 Data Library Conversion

The default-low energy neutron libraries in Geant4, the G4NDL libraries, are given at zero kelvin, which means that any realistic simulation will require significant Doppler broadening. In addition, the data libraries from different simulation codes, such as Geant4 and MCNP, will differ in some cases even if they are both based on the same initial data. The MCNP data libraries, in particular, are based on standard ENDF/B values, but they have been modified to correct deficiencies in the original data [4]. Compounding these issues is the pre-thinning used in the G4NDL v.3.14 libraries, which were the latest version of G4NDL available when this project began.

For these reasons, the MCNP data libraries were converted into the G4NDL format so that the MCNP data could be used in the NStable code. A Python script was created to read the data for a single isotope from the MCNP *endf* libraries, then parse the MCNP format, and finally create the necessary files in the G4NDL format. As noted in Section 2.3.1, the G4NDL data libraries do not have a format manual, so the format was reverse engineered using the Geant4 source code [12].

Chapter 5

NStable Code Verification and Validation

In code development, two processes are used to ensure that a code is functioning properly: verification and validation. Verification is performed to ascertain whether the code is functioning as expected and fulfils the design requirements, whereas validation is performed to ensure that the code is producing *accurate* results [27]. In general, the verification process occurs throughout the code development cycle, and after the code is "complete", it is validated against reputable sources, such as experimental results or other validated codes. In code-to-code comparisons, validation is assumed to be transitive, so that a newly developed simulation code may be validated through its agreement with trusted benchmark codes.

The NStable code was validated against the three neutron transport codes listed in Section 3.1: MCNP5, TART 2005, and DRAGON 3.06J. All of these codes have been validated using code-to-code comparisons, benchmarks, and experimental results [28, 29, 30]. The comparisons shown below covered a range of applications

including criticality calculations and neutron source distributions for finite and infinite geometries. Not every validation code was applied to each comparison problem; in general, the codes were applied to the problems that highlighted their strengths (e.g. DRAGON was only used for near-critical lattice cell calculations).

5.1 Data Libraries

Table 5.1 lists the nuclear data libraries used for each simulation code in the validation study. For DRAGON and TART, the default data libraries were used. In DRAGON, this is the 69 group IAEA library which uses the WIMS-D4 format. This library was created as part of the WIMS Library Upgrade Project (WLUP), and is based on ENDF/B-VII cross sections. Conversely, TART 2005 uses an ENDF/B-VI continuous energy library that was evaluated at 0 K (TART also includes multi-band and multi-group libraries) [24]. To match TART, the endf66 library was chosen for MCNP5, which is also an ENDF/B-VI data library, but it was processed at room temperature (293.6 K). For the NStable code, the default ENDF/B-VII library, G4NDL4.0, was used for comparisons involving DRAGON; otherwise, NStable used the C6-ENDF6 library, which was derived from the MCNP endf66 library using the python script described in Section 4.4.

Table 5.1: List of data libraries used for each code

Software	**Data Library**	**ENDF/B Version**	**Evaluation Temperature (K)**
NStable	G4NDL4.0	7	0.0
	C6-ENDF6	6	293.6
MCNP5	endf66	6	293.6
DRAGON 3.06J	IAEA WIMS-D4	7	Multiple
TART 2005	Continuous Energy	6	0.0

5.2 Simulation Worlds

The NStable code was validated using four main simulation worlds: a pure U235 sphere, a homogeneous UHW sphere, a homogeneous NU lattice cell, and a standard fresh-fuel CANDU 6 lattice cell [18]. In each of these models, one parameter was varied to achieve different levels of criticality. The simulation worlds, and their major features, are outlined in Table 5.2.

All of the simulations were initialized from an angularly isotropic point source of neutrons at the centre of the simulation world, where the energy of the neutrons was based on a Gaussian distribution centred at some arbitrary value. The median energy of these point sources depended on the chosen world configuration. For the lattice cells and the UHW sphere, the energy of the initial neutrons is not important because the world contains moderating elements and/or has periodic boundary conditions. Therefore, the initial neutrons are unlikely to escape the simulation world without being absorbed, so the initial mean kinetic energy was set at 1.0 MeV. However, for the subcritical U235 spheres, fast neutrons easily escaped the sphere without interacting.

Table 5.2: Simulation worlds used in validation cases

Spheres	**U235**	**UHW**
Material	U235 (100%)	U235 (0.077%) U238 (10.653%) D_2O (89.270%)
Radius (cm)	Variable	Variable
Density (g/cm^3)	18.75	3.0143
Temperature (K)	293.6	293.6
Infinite Lattice	**CANDU6 Lattice Cell**	
Material	Fresh fuel	(See reference [18])
Pitch (cm)	28.575	(Variable)
Density (g/cm^3)	10.554 0.8074 1.0888	(Fuel) (Coolant) (Moderator)
Temperature (K)	859.99 561.29 336.16	(Fuel) (Coolant) (Moderator)

In these spheres, only the slow neutrons (approximately 10 eV and lower) were present long enough to undergo fission. Therefore, it was advantageous to set the mean initial energy of the neutrons in this range (0-10 eV). While the source may take longer to converge *in time* (simulation time[1]), the first runs did not experience overwhelming population depletion through escape.

5.3 Simulation Errors

The error reported by nuclear simulation codes is generally only the *statistical error* associated with the simulation. This is the easiest error term to control because it

[1]Simulation time refers to the time tracked by NStable in the simulation world, whereas computation time is the real world time required to run the simulation.

can be minimized by increasing the number of simulations. For an average value, $\bar{x}$,

$$\delta x = \frac{\sigma}{\sqrt{N}} \tag{5.1}$$

where δx is the statistical error of $\bar{x}$, σ is the standard deviation of $\bar{x}$, and N is the total number of x values used in the average. Therefore, increasing the number of discrete samples of x decreases the statistical error inherent in the average $\bar{x}$. However, nuclear simulations also contain errors due to the simulation methods and the discrepancies in the nuclear data. All of the validation codes simulate neutron transport differently than NStable, although in some cases such as TART 2005, these differences are minor. Moreover, the method of calculating the fundamental properties of a system, such as k_{eff}, can also differ (dynamic criticality versus the k-eigenvalue method). In addition, all of the codes use different data libraries, so discrepancies exist between the basic nuclear data that is being sampled in each code. All of these differences will contribute some finite error to any comparisons. Therefore, the statistical error is not expected to account for all of the discrepancies in the criticality or the neutron spatial distributions between the validation codes and NStable, especially since the statistical error can be minimized. To account for these non-statisical errors, an arbitrary limit of 10 mk (0.01) was set for the criticality comparisons; if the criticality estimates from two codes were within 10 mk, these estimates were assumed to agree. This variance of 10 mk is supported by the range of criticality estimates for a CANDU lattice cell using fresh fuel seen in Table 5.6 (see Section 5.7.2, the estimates have a range of 6 mk).

5.4 General NStable Results

Figures 5.1 to 5.3 show the general results of NStable simulations of U235 spheres with respect to time. The two major calculated results, the k_{eff} and Shannon entropy estimates, are shown over the entire simulation, where each data point represents the result of a single run. Each simulation was initialized from a point source at the centre of the simulation world, so the neutron source distribution must converge in space and energy to the fundamental distributions implied by the world definition. This convergence is exhibited in the first N_{sc} runs, where both the Shannon entropy and k_{eff} values are changing with respect to time (N_{sc} is time to convergence of the source in runs). After this initial convergence period, both calculated results stabilize, and are distributed around a constant average value.

The results shown in Figures 5.1 to 5.3 were derived by varying the radii of the U235 spheres. Since the subcritical U235 spheres are dominated by a slower neutron population (lower energy), the source distribution takes longer to converge (in simulation time); this behaviour for subcritical unmoderated systems is also shown by Dr. Cullen in "Static and Dynamic Criticality: Are They Different" [10]. For the sake of efficiency, the run duration should be longer for these subcritical spheres to give the population time to evolve over each run. A short run duration would also be acceptable, although the slower neutrons may not have enough time to undergo any non-scattering interactions. Thus, the statistical accuracy of a short run would be limited for the U235 spheres (e.g. not enough neutron production and loss to properly calculate any physical quantities).

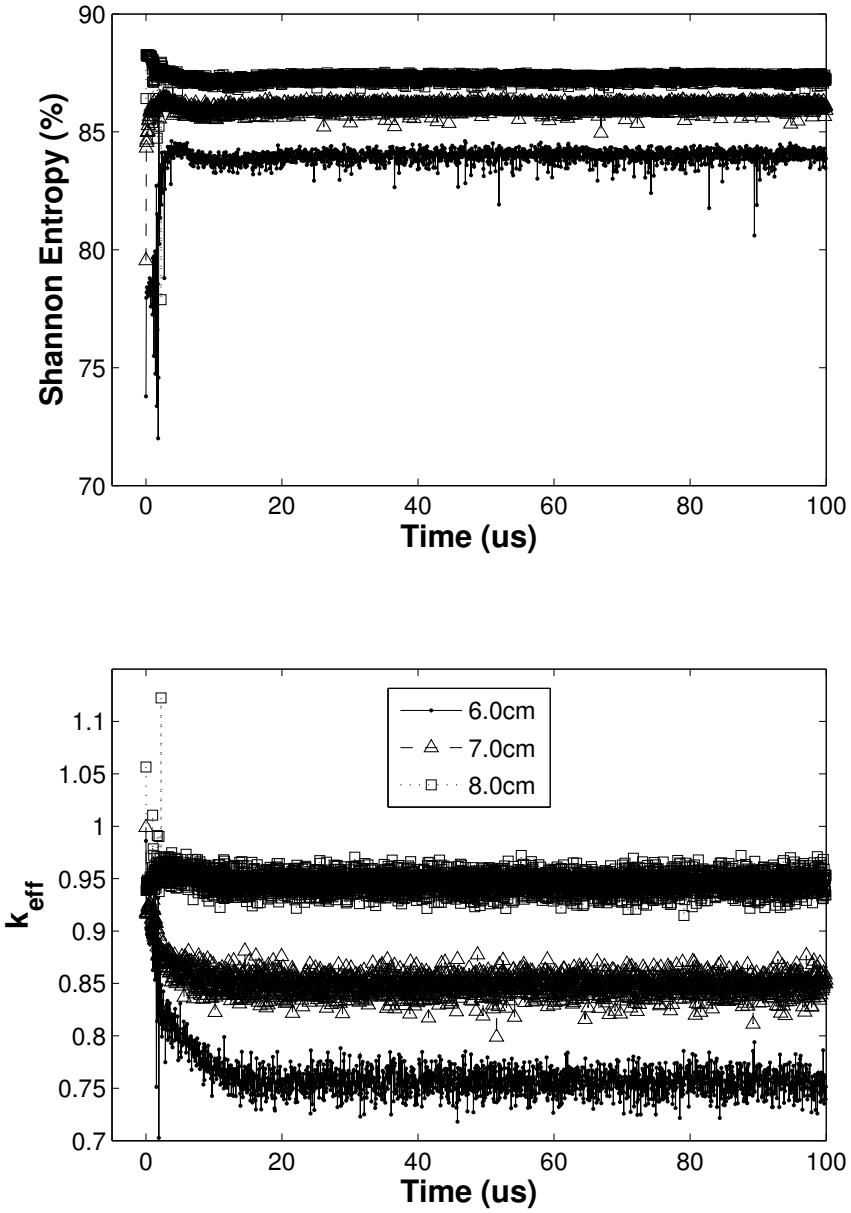


Figure 5.1: Shannon entropy (top) and k_{eff} (bottom) of subcritical U235 spheres.

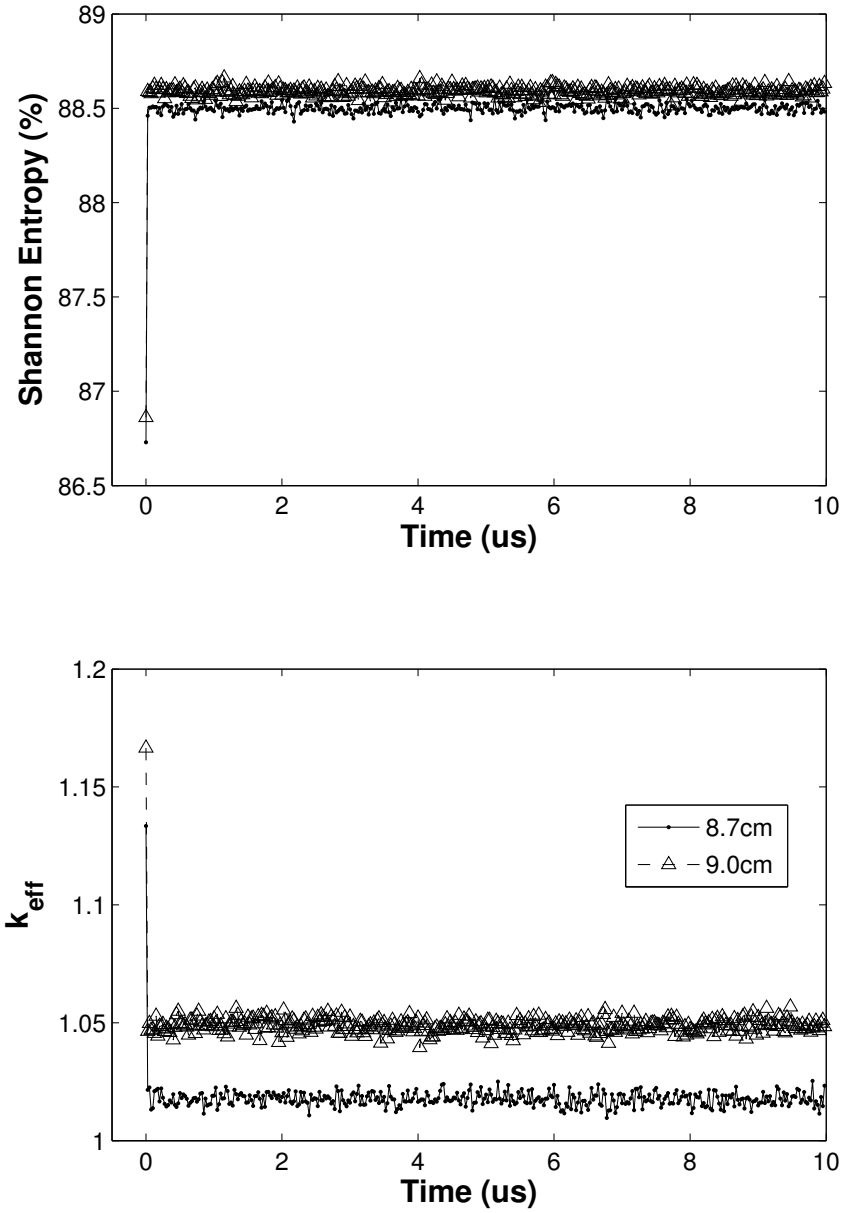


Figure 5.2: Shannon entropy (top) and k_{eff} (bottom) of near-critical U235 spheres.

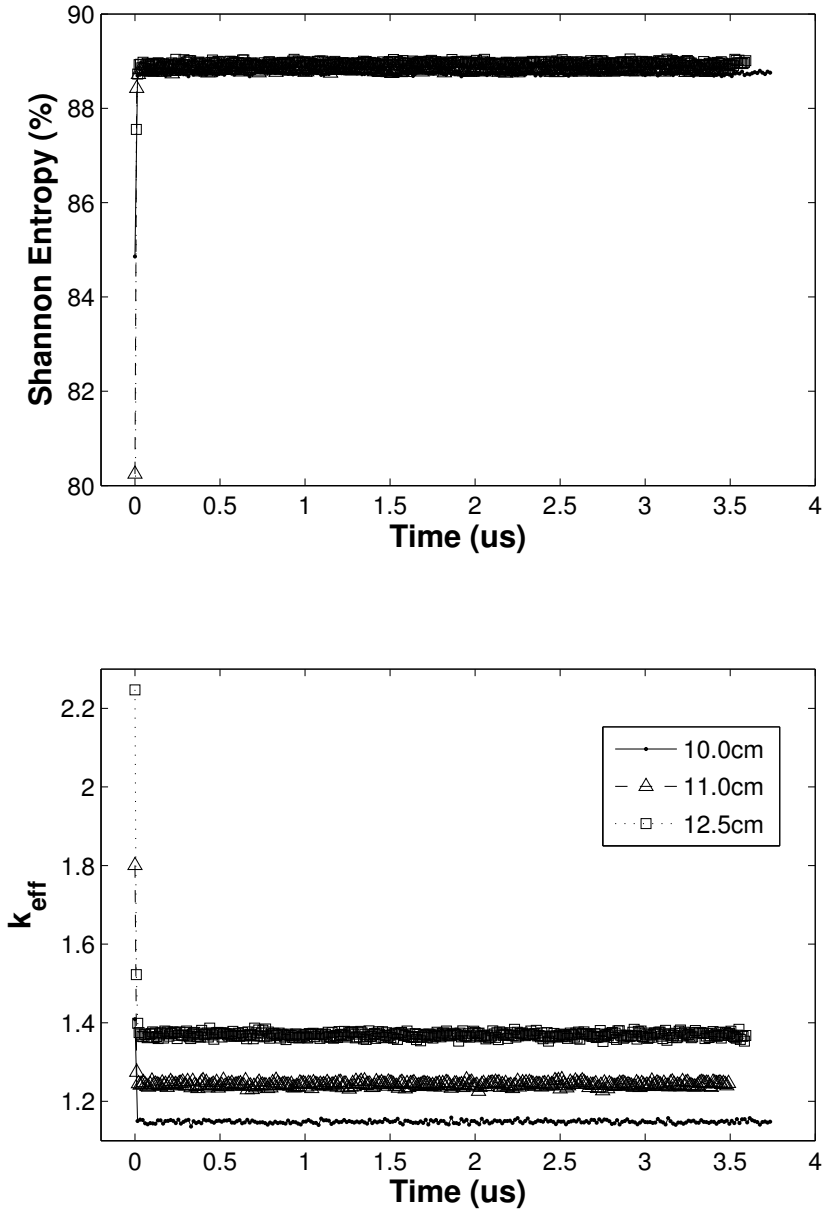


Figure 5.3: Shannon entropy (top) and k_{eff} (bottom) of supercritical U235 spheres.

5.5 Neutron Spatial and Energy Distributions

To accurately calculate the criticality of a system, the NStable code must be able to produce the proper spatial and energy distributions of neutrons. Starting from an arbitrary initial distribution, such as an angularly-isotropic point source, the neutron population in NStable must evolve to the appropriate stable neutron spatial distribution, which depends on the geometry and material composition of the system. First, the NStable code must allow the neutron population to converge to the fundamental spatial and energy distributions, and then it must maintain these distributions in time. Moreover, for accurate criticality calculations, these distributions must match the distributions predicted by other simulation codes.

5.5.1 Source Convergence

Figures 5.1 to 5.3 show that each simulation has an initial period where the Shannon entropy is converging to a stable value. This convergence results from the convergence of the neutron spatial distribution and energy spectrum. Since the simulations started from an angularly-isotropic point source of neutrons with an arbitrary Gaussian distribution of energies, neither the initial energy or spatial distributions were close to the fundamental distributions. Therefore, the population had to evolve over time, and stabilize in both space and energy. The stabilization of a population of one million neutrons is shown in Figure 5.4. The spatial distribution and energy spectrum are shown at six different times since the start of the simulation (simulation time). Up to 50 μs, the neutron source distribution[2] is converging, but by 125 μs, the source

[2]The term *neutron source distribution* refers to the survivors of a run that are saved by NStable and used to initialize the next run. Therefore, the spatial and energy distribution of these neutrons at the end of the current run can be derived from this *source distribution*.

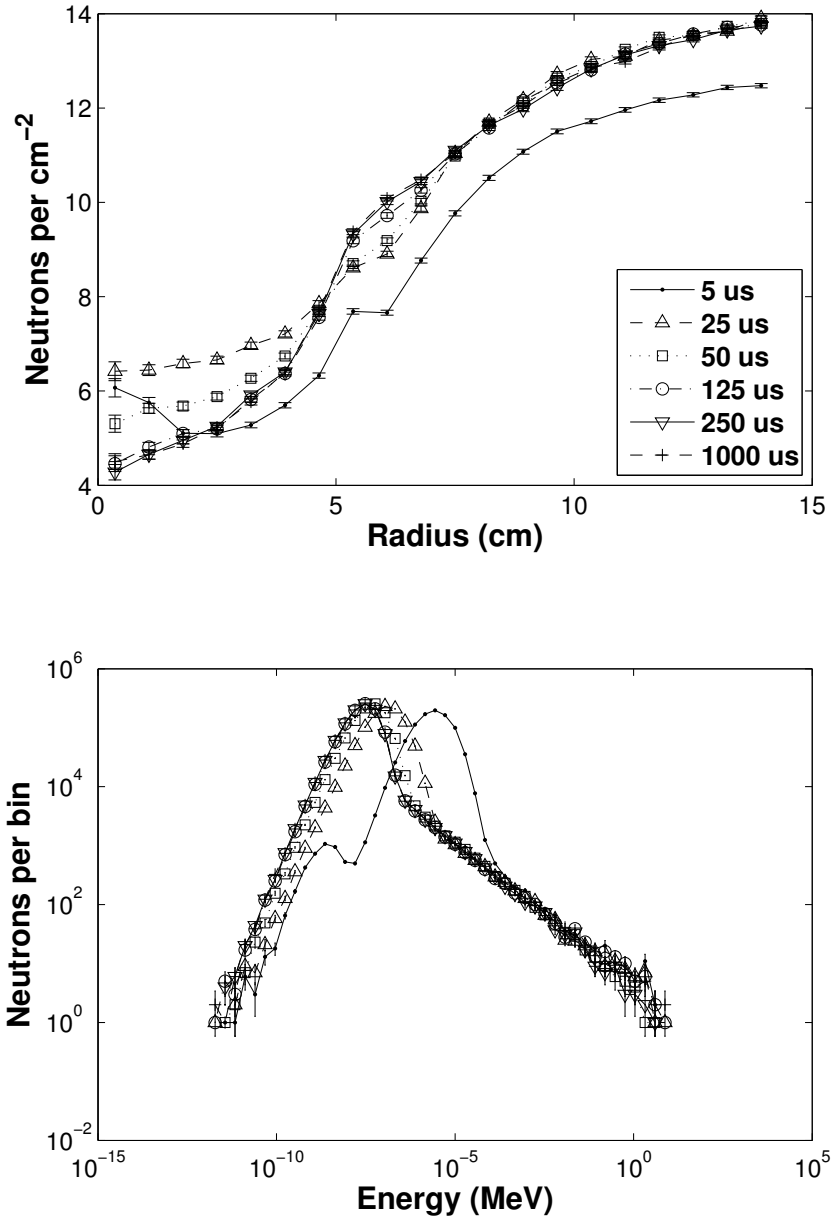


Figure 5.4: Convergence of the neutron spatial distribution (top) and energy spectrum (bottom) of a CANDU 6 lattice cell.

distribution has mostly converged. By 250 μs , the spatial and energy distributions are stable in time, with some variation due to the stochasticity of the simualtion. For simplicity, the delayed neutrons were created instantaneously (at the time of the fission).

5.5.2 Spatial Distribution Comparison

To verify the ability of NStable to reproduce the fundamental neutron spatial distribution, the stable neutron source distribution for a CANDU 6 lattice cell was compared using NStable and DRAGON. This is a useful geometry for this comparison because the CANDU 6 lattice cell is heterogeneous, and therefore, has a more complex spatial distribution. However, DRAGON does not calculate the neutron spatial distribution, or neutron density, directly. Instead, it calculates the neutron flux for each homogeneous region of the spatial discretization and each energy group of the energy discretization (see Section 2.4.1)[3]. The neutron flux (ϕ) is related to the neutron density (n) by the equation

$$\phi = vn \tag{5.2}$$

where v is the velocity of the neutrons, and therefore, varies for each neutron [1]. Since the kinetic energy of the neutrons is discretized in DRAGON, the velocity may be similarly discretized so that the average velocity of an energy group is given by

$$\bar{v}_g = \sqrt{\frac{E^g_{max} + E^g_{min}}{m_n}} \tag{5.3}$$

[3]Flux mapping could be implemented in NStable, but unlike DRAGON it is not required. Building a flux map would require the simulation world to be subdivided many times, which would slow down the simulation.

where $\bar{v}_g$ is the average neutron velocity of group g, E^g_{max} and E^g_{min} are the limits of the energy range spanned by the group, and m_n is the mass of a neutron. Using Equations 5.2 and 5.3, the neutron flux calculated by DRAGON can be converted to a neutron density using

$$n_i = \sum_{g=1}^{N_g} \frac{\phi^i_g}{\bar{v}_g} \tag{5.4}$$

where N_g is the total number of groups in the energy discretization, and i denotes a single homogeneous region in the spatial discretization.

The spatial distribution comparison was performed using the centreline neutron number density in NStable and DRAGON. For this comparison, the centreline region was defined as a 2 cm thick region (slab) in the X-Z plane (the slab was 2 cm thick along the Y-axis, and spanned the lattice cell along the X and Z-axes, See Appendix F.3 for the DRAGON input file). This slab passes through the first and third rings of fuel, but misses the second ring because it is offset by 0.262 radians. With the region of interest defined, the neutrons in this slab could be extracted from an NStable source distribution (snapshot in time), and discretized in the x-direction to match the volumes defined in DRAGON (see Appendix F.3). Since the magnitude of the neutron density is arbitrary for these simulations - in an actual reactor, the magnitude of the density or flux controls the power output of the reactor - the neutron density from one of the codes needed to be scaled. Using a least-squares algorithm, the DRAGON neutron densities were all scaled by a constant factor a such that

$$n'_{dragon} = a\, n_{dragon} \tag{5.5}$$

where n'_{dragon} is the scaled DRAGON neutron densities. The results of this comparison

are shown in Figure 5.5 with units of neutrons/cm^2 because the DRAGON simulation was two dimensional. The hatched regions in the figure denote the four fuel rings (including the centre pin), the pressure tube and the calandria tube respectively (from left to right). Again, the delayed neutrons were created instantaneously.

Multiple source distributions from different times were combined to reduce the stochastic variation of the NStable results (250, 300 and 350 μs). Since all of these snapshots were recorded after the neutron source distribution had converged, they all represent a sampling of the fundamental spatial and energy distribution of the system. For the most part, the comparison is within the stochastic error predicted by the NStable results (the error in the flux predicted by DRAGON was insignificant and is not shown in the figure; the maximum error in the flux was only 0.1%). These discrepancies could be minimized by either using more primary neutrons in the simulations, or by combining more snapshots from different times. Additionally, some discrepancies are expected due to the simplifications used to solve the transport equation in DRAGON, and differences in the nuclear data libraries.

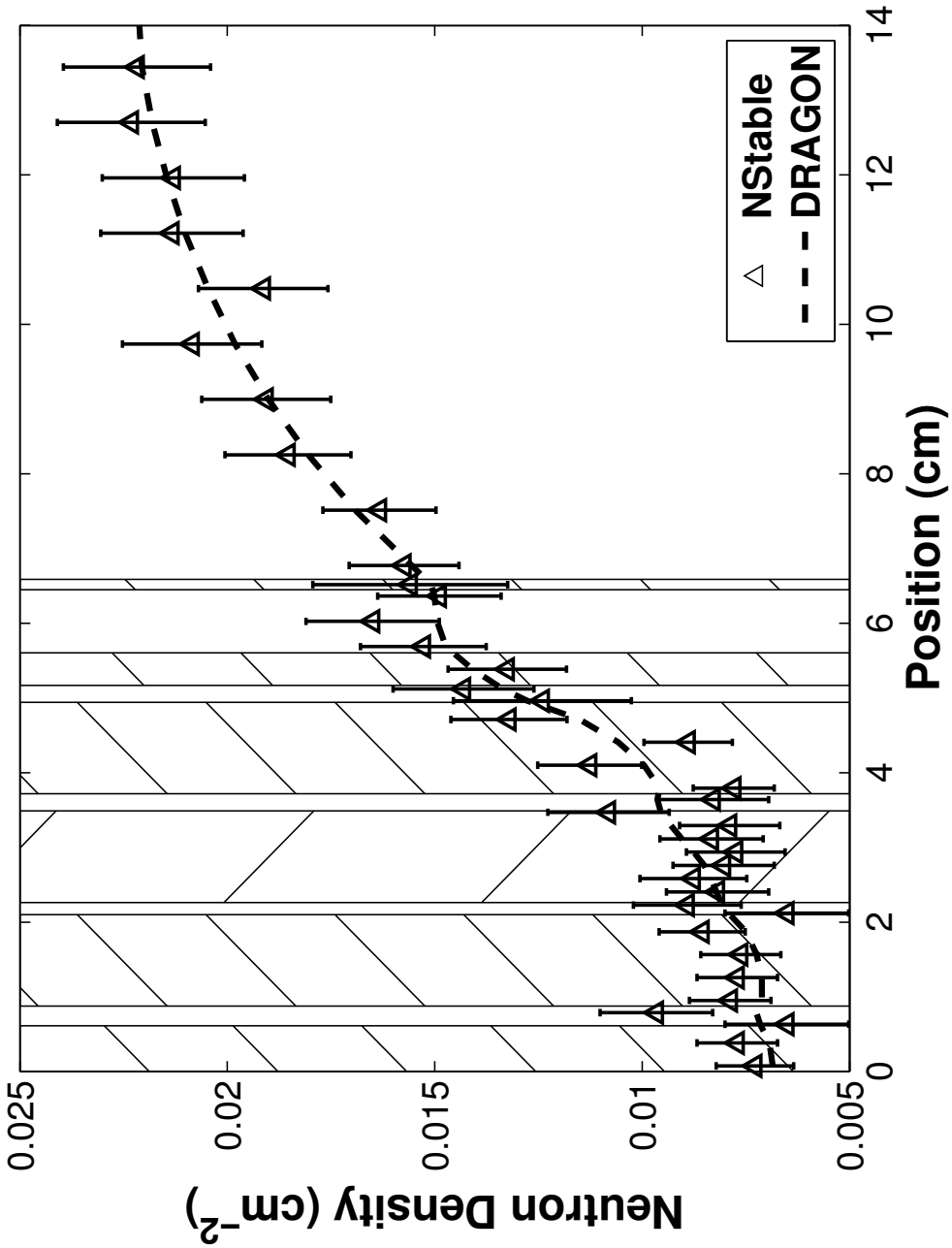


Figure 5.5: Comparison of the predicted centreline neutron spatial distribution in NStable and DRAGON.

5.6 Verification of Unbiased Renormalization

The unbiased renormalization of the neutron population is fundamental to the NStable code. If the renormalization process were biased, then this entire method of population stabilization would be flawed and invalid. The renormalization process used in the NStable code is well known; however, given the importance of this aspect of the code, the accuracy of the renormalization process needed to be shown in terms of preservation of the neutron spatial and energy distributions.

To compare the effect of renormalizing the neutron population, three U235 spheres (one subcritical, one near-critical and one supercritical) were simulated over the initial period of source convergence both with and without renormalization. All of the simulations started from an angularly isotropic point source of neutrons at the centre of the spheres with an average energy of 1 MeV. Given that the neutron population changes quickly without renormalization when $k_{eff} \neq 1$, these simulations were performed over 50 or fewer runs with spheres that were at most 30 mk from critical. Since the simulations were extremely short, delayed neutrons were not included. The details of the three pairs of simulations are summarized in Table 5.3.

Table 5.3: Renormalization Simulations

	Subcritical	**Near-Critical**	**Supercritical**
Radius (cm)	8.2	8.5	8.7
Runs	30	50	30
k_{eff}	0.973	1.008	1.030
Initial Population	2.0e+6	1.0e+5	1.0e+4
Final Population	3.8e+4	1.7e+6	1.0e+6
Initial Pop. (renorm)	2.0e+5	1.0e+5	2.0e+5
Final Pop. (renorm)	1.7e+5	1.1e+5	2.3e+5

Each pair of simulations was identical except for the number of initial particles and whether or not renormalization was used. Therefore, the spatial and energy distributions at the end of each simulation should be identical to its pair, within error. Figures 5.6 to 5.8 show the comparison of the one-dimensional spatial (radial) and energy distributions for each pair, where the final neutron population of each renormalized simulation was scaled to match the final number of neutrons in the complementary simulation without renormalization. The spatial distributions are shown in terms of linear and volumetric radial neutron densities, where the volumetric radial density is calculated as neutrons per cm^3 along the radius of the sphere.

In every case, the agreement between each pair of simulations breaks down for bins with relatively few neutrons. For example, the volumetric spatial distribution shows worse agreement near 0 cm because the volume of this region is relatively small, and thus contains few neutrons. Therefore, any discrepancies are magnified. Likewise, the ends of the energy spectrum show worse agreement because of the relatively low number of neutrons at these energies. These differences are quantified in Figure 5.9, which shows the discrepancies between the renormalized and not renormalized populations that *cannot* be accounted for by the statistical error. For the spatial distribution, the difference peaks at 2.5% when the sphere radius is under 10 mm. In the energy spectrum, the percent difference exceeds 10% for the low energy neutrons in the 8.5 and 8.7 cm spheres. However, the total number of low energy neutrons in each bin is at least three orders of magnitude less than that of the most populous bins. Therefore, the discrepancies are the result of the small sample size; a discrepancy of tens of neutrons is not very significant in a population of millions. These errors could be reduced by increasing the number of neutrons simulated, or by more closely matching

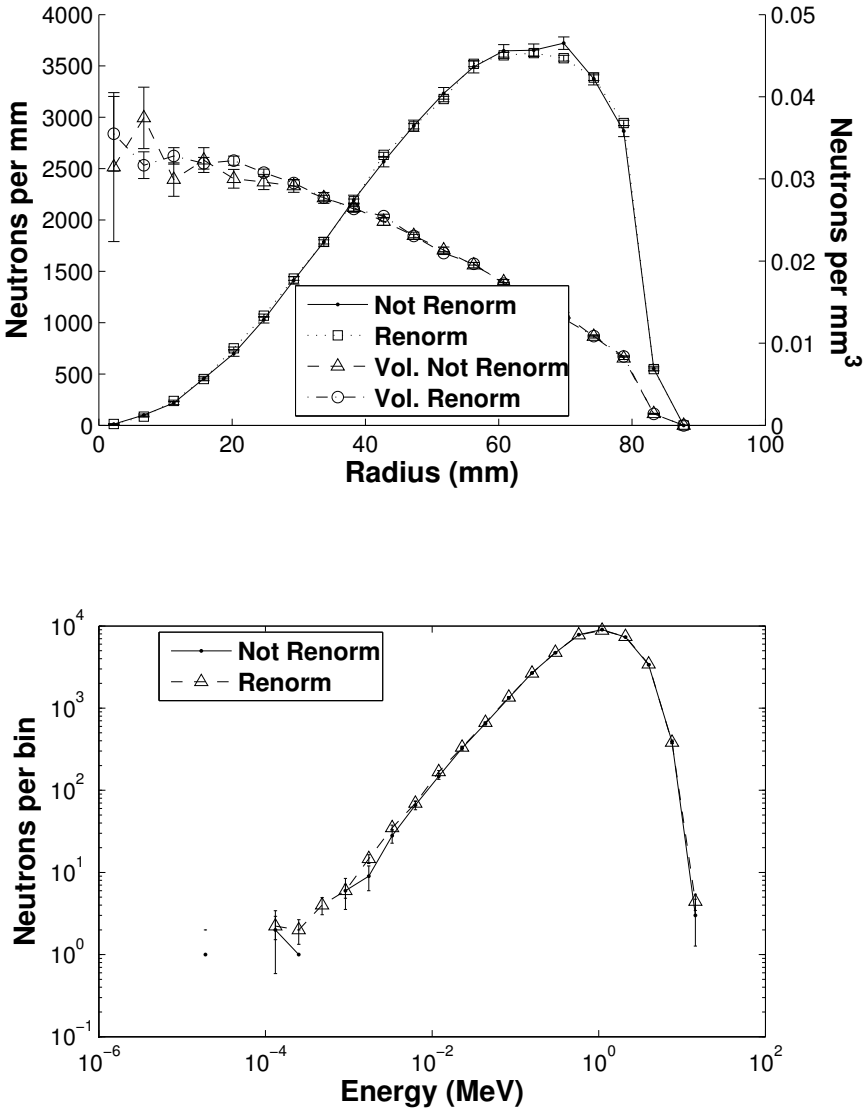


Figure 5.6: Comparison of spatial and energy distributions for a renormalized and not renormalizaed neutron population in a subcritical 8.2 cm U235 sphere.

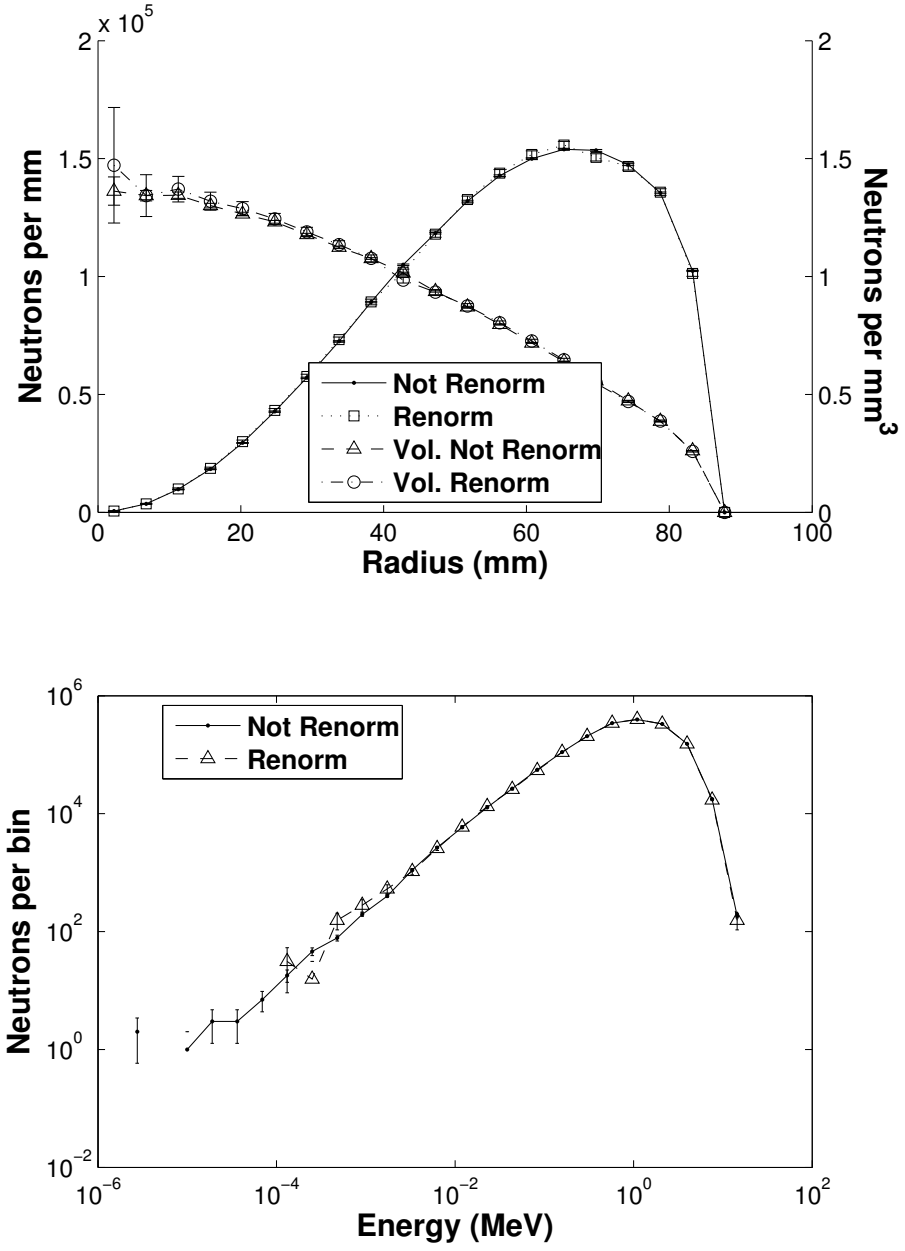


Figure 5.7: Comparison of spatial and energy distributions for a renormalized and not renormalizaed neutron population in a near-critical 8.5 cm U235 sphere.

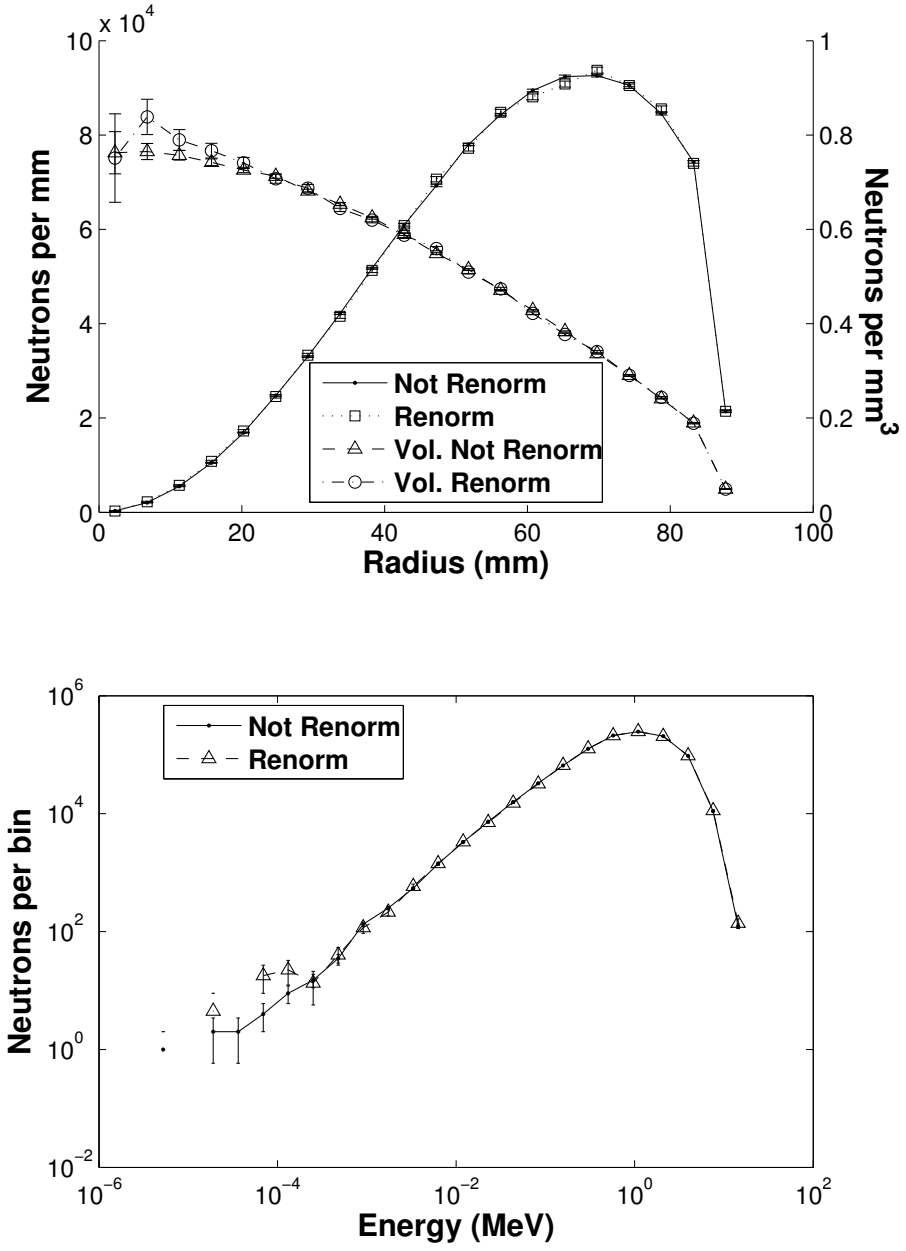


Figure 5.8: Comparison of spatial and energy distributions for a renormalized and not renormalizaed neutron population in a supercritical 8.7 cm U235 sphere.

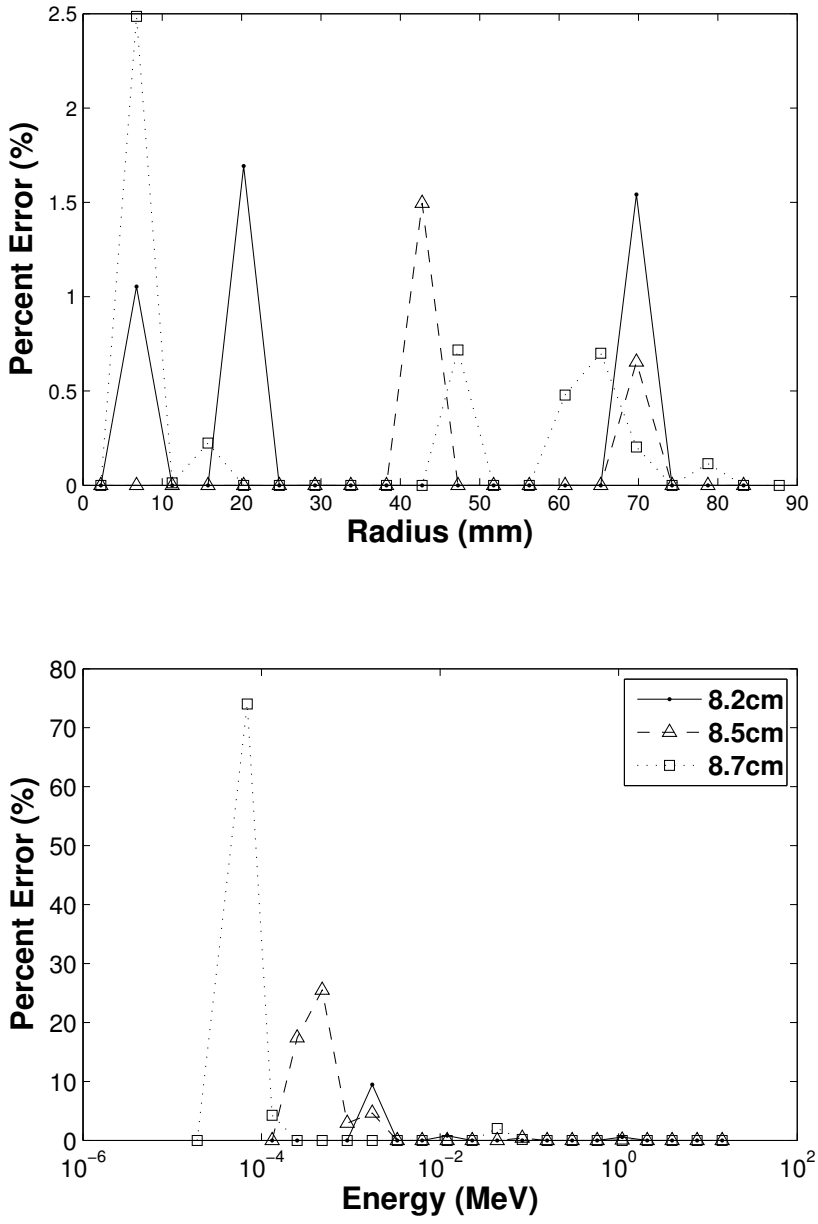


Figure 5.9: Percent error in the renormalized and not renormalized spatial and energy distributions.

the final neutron populations of each pair of simulations so that less scaling is necessary in the analysis. Thus, the renormalization process is unbiased and does not affect the spatial or energy distributions of the neutron population.

5.7 Criticality Calculations

Since criticality depends both on the composition of the simulation world and the neutron source distribution, it is a useful quantity to compare for code validation. Any discrepancies in the simulation of the neutron population should affect the criticality estimate, k_{eff}. Thus, the NStable code was compared to all three validation codes in terms of the criticality of various simulation worlds.

5.7.1 Finite Geometries

In the finite geometries, the U235 and UHW spheres, the criticality estimates of NStable were compared to the validation codes for spheres of varying radii. The main comparison code was TART 2005 because it can accurately simulate sub- and supercritical systems [10]. The NStable results were also compared to MCNP, where the MCNP results were produced using k-eigenvalue calculations. Results were produced using MCNP for every simulation, but they were most applicable for the near-critical systems. In all of the finite geometry simulations, ENDF/B-VI data was used for the best comparison with TART 2005. Additionally, delayed neutrons *were not simulated*.

The average criticality estimates for each sphere are shown in Figures 5.10 and 5.11. The averages in TART 2005 and NStable were calculated using the last 25% of the runs, while the average MCNP k_{eff} estimates were calculated over the last 225 runs.

Table 5.4: Finite Geometry Criticality Simulations

U235 Spheres	**NStable**	**TART**	**MCNP**
Initial Population	1e+5	2.5e+4	5e+3
Runs	250 - 4000	1000 - 4600	250
Run Duration (ns)	8 - 75	6 - 250	N/A
UHW Spheres	**NStable**	**TART**	**MCNP**
Initial Population	1e+5	2.5e+4	5e+3
Runs	350	200 - 280	250
Run Duration (μs)	100	300	N/A

Additionally, the error bars represent the *statistical* error in the average k_{eff} estimates for each code. In the case of MCNP and TART, variance reduction techniques are built into these averages; each calculated k_{eff} value in MCNP is itself an average of three different k_{eff} estimators [4]. For all three simulations, the statistical error was less than 1 mk for every k_{eff} estimate.

These comparisons show excellent agreement between TART and NStable at every point; the error is consistently less than 10 mk in absolute value. As expected, the MCNP results do not show agreement except in the near-critical region. Outside of this region, MCNP underestimates the criticality of the spheres. Additionally, the criticality of the UHW spheres appears to be approaching an asymptote in the supercritical region. Since the largest UHW sphere is almost 2.8 metres in diameter, the escape probability of a neutron in the sphere is low, so the criticality estimates are approaching that of an infinite lattice (k_{inf}).

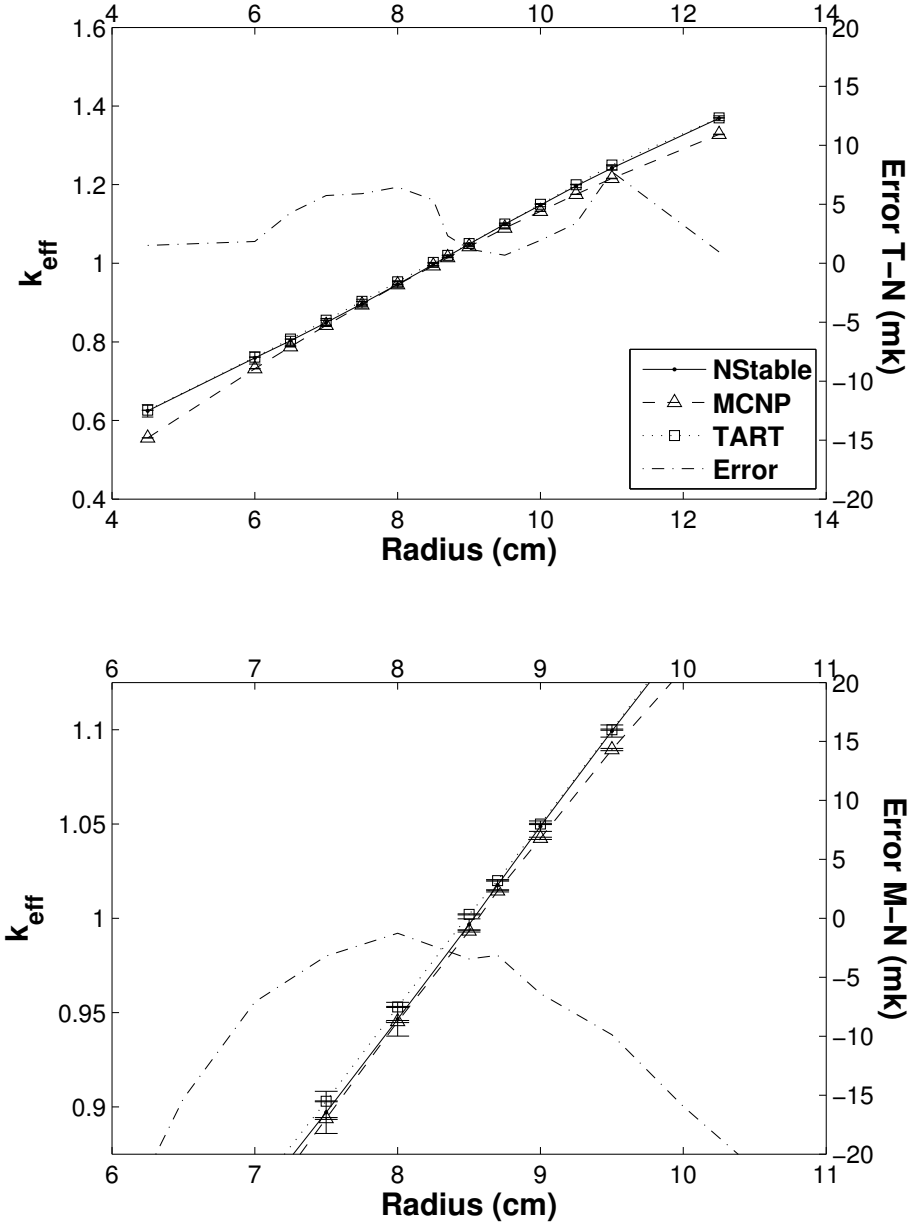


Figure 5.10: Criticality estimates for U235 spheres of varying radii using NStable, TART and MCNP (top). The same estimates are also shown for the near-critical region only (bottom).

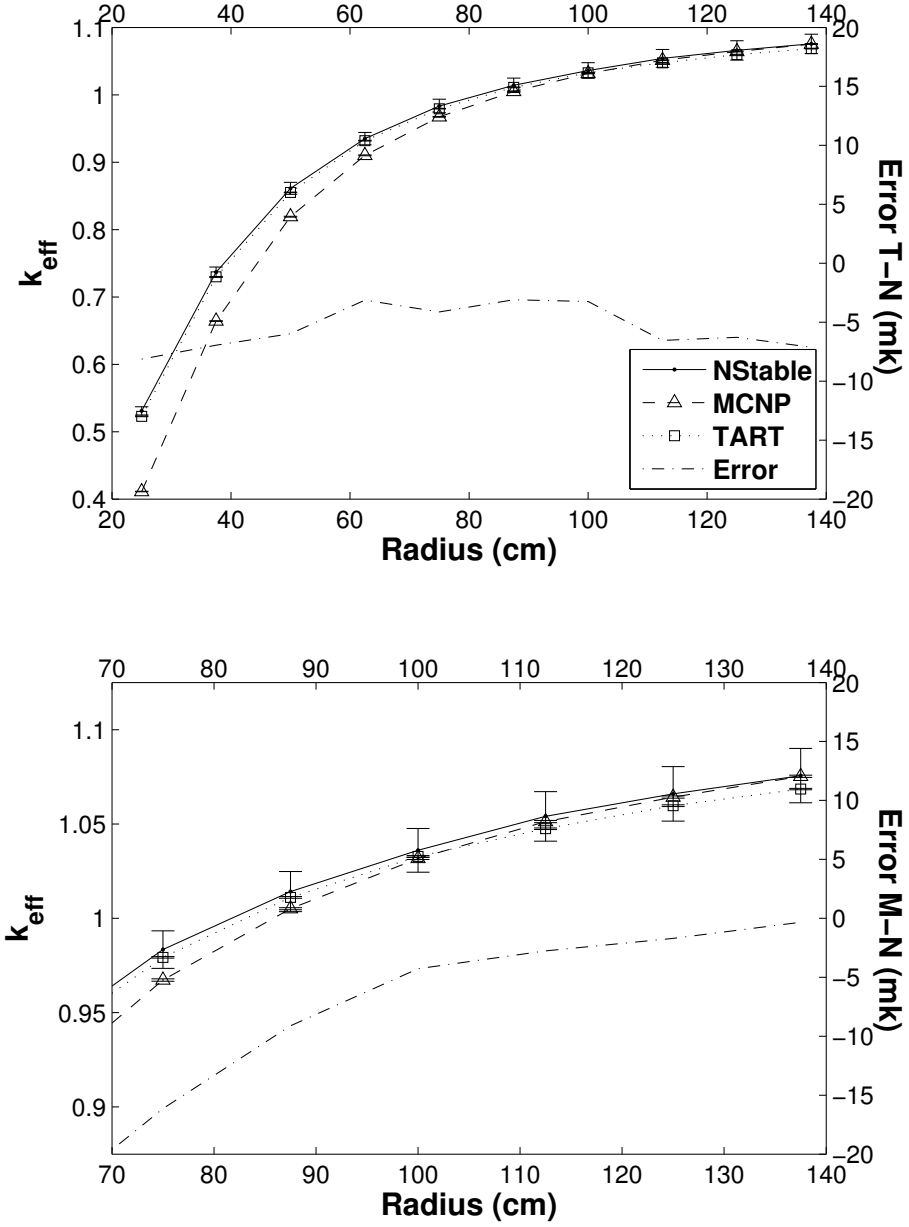


Figure 5.11: Criticality estimates for UHW spheres of varying radii using NStable, TART and MCNP (top). The same estimates are also shown for the near-critical region only (bottom).

5.7.2 Infinite Geometries

Similar comparisons were carried out for infinite geometries, namely infinite lattices of the CANDU 6 lattice cell simulated at various lattice pitches using NStable and DRAGON. For these simulations, the delayed neutrons were produced instantaneously (i.e. $\nu = \nu_{total}$). The specific simulation parameters are given in Table 5.5.

Table 5.5: Infinite Geometry Criticality Simulations

CANDU 6 Lattice Cell	**NStable**	**DRAGON**
Initial Population	1e+5	N/A
Runs	150	N/A
Run Duration (μs)	100	N/A

Figure 5.12 shows the criticality estimate comparisons for the infinite geometries. Both show reasonable agreement with the comparison code, where the error is less than 10 mk in all cases. Agreement in the CANDU 6 lattice cell shows that NStable can handle heterogeneous systems. While there is some discrepancy between the NStable code and DRAGON, the NStable estimate at the standard lattice pitch (28.575 cm) lies between the DRAGON estimate and those made by other codes. Table 5.6 shows the results of two codes from the SCALE package, along with the DRAGON and NStable estimates [22]. All of the estimates are within 6 mk.

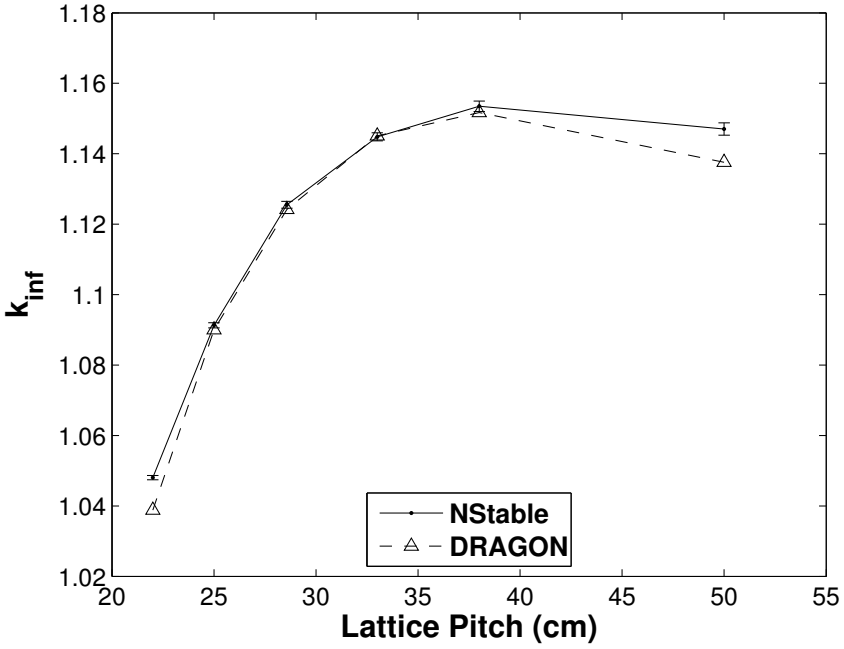


Figure 5.12: Comparison of criticality estimates for a CANDU 6 lattice cell with varied lattice pitches.

Table 5.6: Comparison of criticality estimates for a CANDU 6 lattice cell

Simulation Code	k_{eff}	**Notes**
NStable	1.127	Monte-Carlo, continuous energy
DRAGON	1.124	Deterministic, 69 energy groups
SCALE/NEWT	1.129	Deterministic, 238 energy groups
SCALE/KENO-VI	1.130	Monte-Carlo, continuous energy

5.8 Transient Simulations

To show the ability of the NStable code to model transient behaviour, a simple transient simulation was performed by varying the temperature of the 87.5 cm radius UHW sphere with respect to time. The temperature was increased linearly from 293.6 K to 1000 K over a period of 50 ms. To ensure that the source convergence did not add additional effects to the simulated transient, the neutron distribution had to be stable before the transient could begin. To ensure this precondition is met, the run manager in NStable waits until the Shannon entropy of the neutron source distribution has converged before beginning any material or geometry transients. Once the source distribution converged, the transient was initiated by incrementally changing the temperature of the homogeneous UHW material (see Section 4.2.3.1). The specific parameters used in the transient simulation are given in Table 5.7. Since this is only supposed to show the ability of the NStable code to model such transients, the transient is not based on any real experiments and delayed neutrons were not included.

Table 5.7: Transient simulation parameters

Simulation Parameter	**Value**
Material	UHW
Radius (cm)	87.5
Run Duration (μs)	100
Initial Temperature (K)	293.6
Final Temperature (K)	1000.0
Time at Initial Temperature (ms)	10
Interval of Change (ms)	50
Time at Final Temperature (ms)	20

The resultant transient is shown in Figure 5.13. Initially, k_{eff} is changing due to

the convergence of the source distribution; this occurs before the transient begins. Once the Shannon entropy has converged, the temperature is held at 293.6 K for 10 ms, producing the initial plateau. Then as the temperature increases, k_{eff} value decreases, before reaching a second plateau after the temperature variation ends. For this simulation, there should be some lag between the temperature variation and the k_{eff} estimate. However, given that the runs were 100 μs long, this is sufficient time for the neutron population to react to small changes (i.e. to be upscattered by the warmer environment). With a shorter run duration, some the simulation should exhibit some lag.

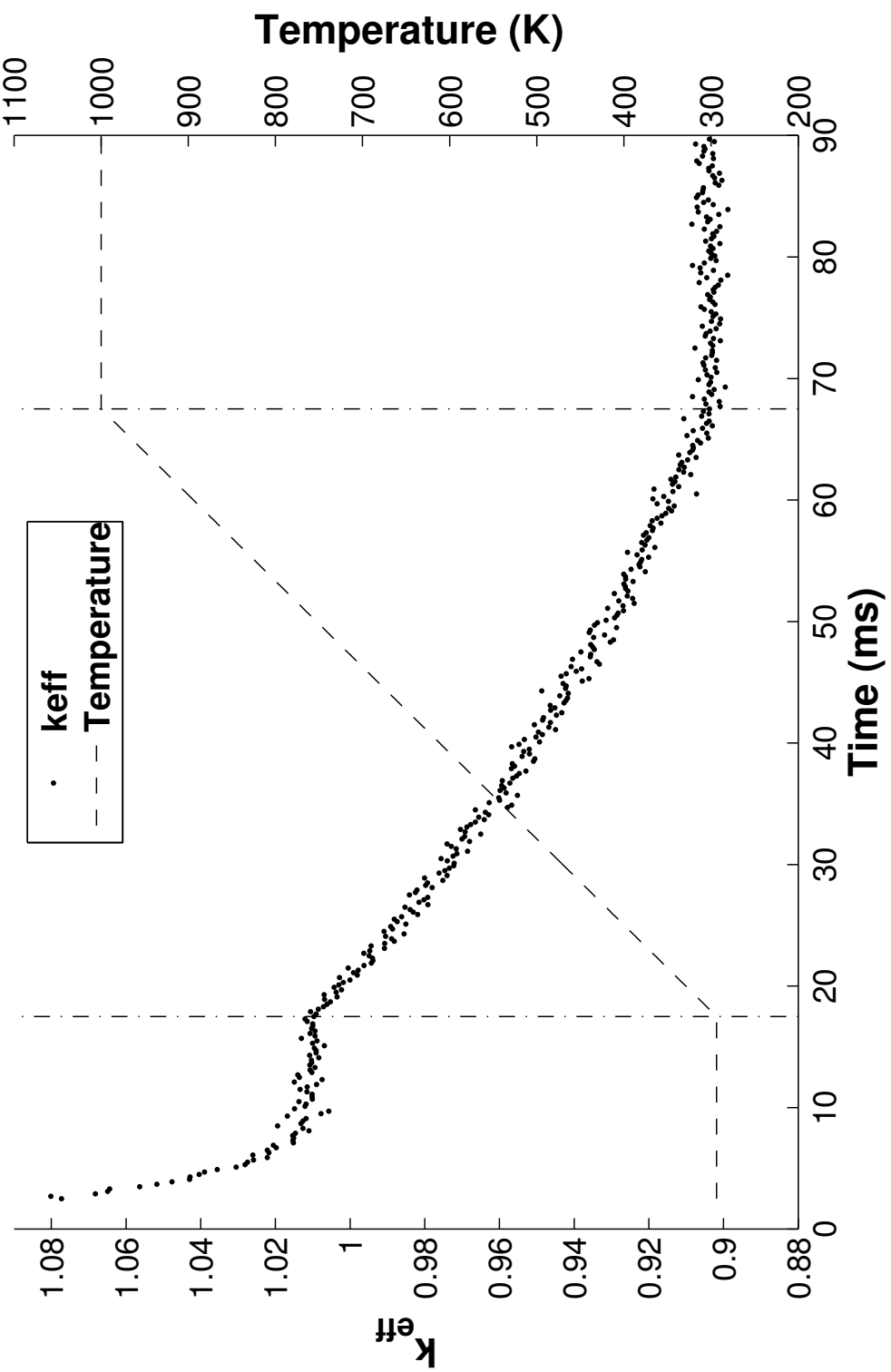


Figure 5.13: Transient for a 87.5 cm radius UHW sphere where the temperature rises from 293.6 to 1000 K (delayed neutrons not simulated).

5.9 Parallelization Gain

Event-level parallelism uses a master-slave architecture to speed up the computation of the events. The time taken by the slaves to process N_E events can be reduced by adding more slaves, assuming the number of events exceeds the number of slaves. If the slaves exceed the number of events, then the excess slaves will not have any events to process, and they will be idle. However, some actions must always be taken by the master. In NStable, the master is not only responsible for coordinating the slaves, but it also calculates all run-level results (e.g. k_{eff}, Shannon entropy). Thus, each run has two components, the event processing by the slaves, which takes T_E seconds, and the run processing by the master, which takes T_M seconds. Since the events are completely independent, the total time taken for a run can be given by

$$T_{run} = \frac{T_E}{N_s} + T_M \tag{5.6}$$

where T_{run} is the total time taken by a run, T_E is the time to process all N_E events, T_M is the time taken by the master to set up and analyse the run, and N_s is the number of slaves available. Note that Equation 5.6 is simplified because it does not account for the additional time needed to coordinate more slaves.

Since the master needs time to analyse a run and set up the next one, event-level parallelism does not have a perfect one-to-one gain (speedup) for the number of processors used. Thus, it is important to measure the efficiency of the parallelization of the code. The gain for a parallel process is defined as

$$G = \frac{T_{run}(1)}{T_{run}(N_s)} = \frac{1}{\frac{1-\alpha}{N_s} + \alpha} \tag{5.7}$$

where α is the non-parallelizable part of the code, $\alpha = T_M/T_{run}$, and $(T_M + T_E)/T_{run} = \alpha + (1-\alpha)$. Equation 5.7 is known as Amdahl's Law, which also predicts a maximum gain [31]

$$G_{max} = \frac{1}{\alpha} \tag{5.8}$$

The parallel processing gain for the NStable code was determined by simulating a CANDU 6 lattice cell over 100 runs with different numbers of slaves. Each simulation was performed using two events per slave and approximately 100,000 primary neutrons (± 20). The average times to simulate a run are shown in Figure 5.14, along with the initialization time of each simulation.

The initialization time of the simulation is the time needed to load the NStable program and the nuclear data before any calculations can begin. Unsurprisingly, the initialization time of the simulation increases as the number slaves increase, but only by approximately 25 seconds for 30 slaves (relative to the minimum initialization time). More importantly, the simulated gain matches closely with the fit using Amdahl's law. From the fit, α is 0.348% for this problem, which implies that the overwhelming majority of NStable is parallelizable for a representative problem. Note that the simulated gain could be increased by increasing the computation time of each event (e.g. by increasing the number or primary neutrons), so the actual gain achieved is problem specific. Likewise, the simulated gain at thirty processors was 27.4, showing only a small loss from the ideal one-to-one gain. Additionally, Amdahl's law predicts a maximum gain of 284 (with infinite slaves), so the NStable code can run more efficiently on more than 30 processors.

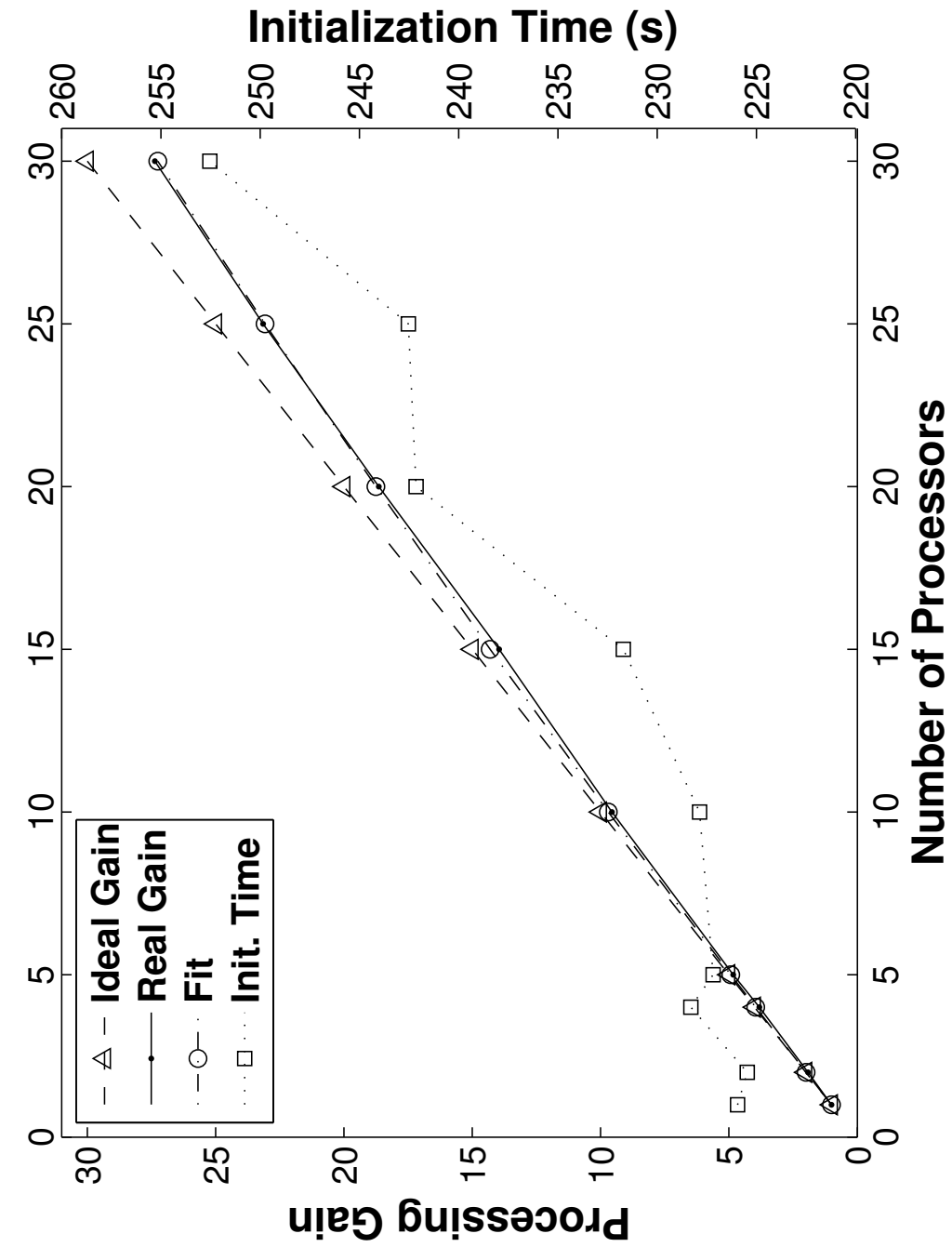


Figure 5.14: Parallel processing gain for NStable where the actual gain is fitted with Amdahl's Law. The plot also shows the initialization time per number of processors.

Chapter 6

Conclusions

Most current dynamic nuclear simulation codes rely on simplifications or quasistatic approximations to model transient behaviour. One common approach, the point kinetics approximation, assumes that the flux shape of the system is constant, and approximates changes in the flux amplitude (neutron population) as a time dependent gain that multiplies the constant flux shape. This is a valuable technique, but it breaks down when the spatial distribution of the neutrons is time-dependent. Other approaches solve for static system parameters, such as the neutron flux and k_{eff}, and then move the simulation world forward in time using isotopic evolution calculations, such as the Bateman equations, before solving for the reactor parameters at the next time step. This coupled quasistatic approach is most often used to calculate the burnup of reactor fuel over months or years. However, it does not capture the short-lived transient behaviour often seen in reactor accidents because the reactor parameters are always calculated statically. Therefore, this project sought to develop a stochastic Monte Carlo code capable of modelling the evolution of a neutron population on small time scales (less than ten seconds).

The objective of this thesis was to create a reactor simulation code capable of following a neutron population in time and space, while also allowing for time-dependent changes in the simulation world. Such a code could be used to calculate important reactor quantities, such as k_{eff}, and also to simulate the reactor transients seen in accident scenarios. The Geant4 simulation toolkit was chosen as the base for this project because of its flexible, adaptable and open-source architecture. While it is not inherently designed to perform reactor simulations (i.e. simulations of a neutron population over time periods on the order of seconds), it is designed to be extended by the user without having to modify the base source code. Geant4 provides basic physics models, geometric descriptions and tracking algorithms, which the user builds from to create a full simulation code. Most other particle physics codes are not as open, and restrict the user to a set of applications defined by the code developers (unless the user is willing to modify the source code). Therefore, Geant4 proved to be a valuable tool for this project.

The resulting code, NStable, was built from the Geant4 toolkit and satisfies the project objectives. The NStable code builds on the Monte Carlo knowledge that has been gained over the past forty years, especially from established codes such as MCNP or TART. However, the flexibility of Geant4 allows NStable to pick the best attributes of these codes, as well as add additional functionality like dynamic materials. In particular, the NStable code can follow a neutron population in time for any given timespan, assuming the user is willing to run the simulation for the necessary length of time. This is accomplished by periodically renormalizing the neutron population without bias. This renormalization keeps the population at a manageable size regardless of the sub- or supercriticality of the simulation world.

During these breaks in the simulation, when the neutron population is renormalized, the material or geometric composition of the simulation world may also be changed. The incremental change in these properties allows the NStable code to approximate the continuous change in the simulation world that is seen in reactor transients.

NStable is able to calculate a variety of system characteristics from the evolution of the neutron population. Foremost among these characteristics are the Shannon entropy, which can be used to determine the stability of the neutron spatial distribution, and the reactor multiplication constant, k_{eff}, which is a measure of the stability of the neutron population with respect to time. In particular, k_{eff} is very important because it relates directly to the controllability of the simulated system; if $k_{eff} > 1$, then the neutron population is growing exponentially, as is the power that it is producing through fission. This state, if left unchecked, can cause an accident, such as melting of the reactor fuel. The NStable code is also able to calculate other characteristics such as the average neutron lifetime in the system, and the total production and loss of neutrons per run.

The NStable code also features a series of optional input arguments designed to extend the applicability of the code. These features include periodic boundary conditions, which can be used to simulate an infinite lattice of fundamental cells, instantaneous delayed neutrons, where the delayed neutrons are born at the same time as the prompt neutrons but are still created using the delayed neutron energy and angular distributions, and unnormalized simulations, which can be used to check the renormalization bias (which should not exist). Additionally, the NStable code currently features three different simulation geometries: a simple homogeneous sphere of varying materials, a homogeneous lattice cell that is the infinite analogue of the

sphere, and a CANDU 6 lattice cell (fresh fuel). Moreover, these simulation worlds are built using a standardized world constructor (*NSWorldConstructor*) that can be used as a template to add any desired simulation world to the NStable code.

6.1 NStable Verification and Validation

The accuracy of the NStable code was evaluated against three established, industry standard codes: MCNP5, DRAGON 3.06J and TART 2005. Each of these codes has a particular area of applicability, and was compared to NStable for simulations that best highlighted strengths of each code. MCNP5 is a standard particle physics simulation code and provides accurate eigenvalue calculations (KCODE) for near-critical systems. DRAGON is a deterministic lattice cell code, and excels at near-critical simulations of complicated, heterogeneous systems such as the CANDU 6 lattice cell. Finally, TART 2005 is a dynamic, time dependent Monte Carlo simulation code and is the most similar to the NStable code developed in this thesis. Like NStable, it can accurately simulate neutron populations in sub- or supercritical mediums through renormalizing the population at regular intervals.

First, the renormalization algorithm was checked to ensure that it was unbiased. This was confirmed by simulating near-critical U235 spheres over the initial period of source convergence both with and without renormalization. As shown in Figures 5.6 to 5.8 and 5.9, the normalized and unnormalized energy and spatial distributions exhibit the same shape, although discrepancies begin to appear in regions with relatively low numbers of neutrons. The percent error difference between the two cases tended to be less than 5%, except in the outlier regions (which have less neutrons). Presumably, these discrepancies should disappear if the simulations were repeated

with more neutrons because the renormalization algorithm used in this project is closely based on well established combing (population control) algorithms [10].

After unbiased renormalization was established, the NStable code was compared to the validation codes using criticality estimates. These are indicative of the accuracy of the simulation since k_{eff} depends on the neutron distribution in time and space, as well as the composition of the simulation world. Thus, any discrepancies in the simulation should appear in the k_{eff} estimate. Criticality estimates from NStable were compared against TART and MCNP for spheres of U235 and UHW (see Figures 5.10 and 5.11). As expected, NStable agreed with TART within 10 mk for the entirety of both simulations, and agreed with MCNP in the near-critical regions (approximately $k_{eff} = 1.0 \pm 50$ mk). For the CANDU 6 lattice cell, the NStable k_{eff} estimates agreed with DRAGON within 10 mk when varying the lattice pitch of the CANDU 6 cell from 22 cm to 50 cm. Thus, the NStable code shows good agreement for criticality calculations across a variety of simulation conditions.

The neutron spatial distribution calculated by NStable was also compared to DRAGON for the CANDU 6 lattice cell. Given the heterogeneity of this simulation world, accurately reproducing the spatial distributions of neutrons is very important. The centreline neutron density of the lattice cell from DRAGON was compared to the spatial distribution of neutrons in NStable at a snapshot in time. The DRAGON neutron densities were scaled by a constant factor to match the NStable densities, and the results are shown in Figure 5.5. Both distributions exhibit the same basic shape, although the NStable data is far noisier. This was expected given that the NStable code uses stochastic calculations, and the statistical discrepancies could be eliminated by initializing more primary neutrons.

6.2 Transient Simulation

A simple transient simulation was modelled by incrementally changing the temperature of the 87.5 cm UHW sphere. The temperature was increased linearly from 293.6 K to 1000 K over a period of 50 ms (100 μs run duration). In response, the k_{eff} value decreased by approximately 100 mk over the same period. The results in Figure 5.13 appear to exhibit a slight lag between the temperature increase and the k_{eff} decrease, but this may simply be stochastic error. Furthermore, the run duration was likely too long to capture any really fine transient behaviour (100 μs run duration). Regardless, the NStable code has been shown to produce transient behaviour in the macroscopic properties of the system (e.g. k_{eff}) in response to a change in the simulation world.

6.3 Parallelization of NStable

Since Monte Carlo stochastic calculations are computationally intensive, spreading the load across multiple processors can greatly reduce the simulation time. Since the NStable code uses event-level parellelism, the master must coordinate the slaves and perform any run-level analysis. Therefore, the gain in computer power (speedup) due to adding additional slaves is defined by Amdahl's law [31]. This relation is shown in Figure 5.14, where Amdahl's law was fit to the actual gain achieved for simulations of a CANDU 6 lattice with varying numbers of slaves. This showed that for a CANDU 6 simulation of 100,000 primary neutrons and 100 μs runs, the unparallelizable fraction of the simulation was only 0.35% in terms of computation time. Therefore, the NStable code has a nearly one-to-one gain in terms of processing time when calculating this problem with slave processors; it achieved a gain of 27.4

when using 30 slaves (the ideal one-to-one gain would be 30).

6.4 Future Work

Given the adaptability of Geant4 and the relative infancy of the NStable code, there is ample opportunity to improve and extend it. This thesis focused on creating the basic functionality necessary to track neutron populations in time, and laid the groundwork for future simulations of transient reactor behaviour. More importantly, this code can now be used as a research tool to better understand reactor transients.

6.4.1 Transient Validation and Modelling

The NStable code has the ability to model transient behaviour in a reactor through dynamic material or geometric properties and a continuous time evolution of the neutron population. However, this potential has never been fully investigated or utilized. Therefore, one of the first tasks should be to attempt to model an actual transient where the material or geometric parameters are well known. For example, the NStable code could be validated against experimental or simulation data for transients involving either the coolant void reactivity or fuel temperature feedback effects in a CANDU 6 lattice cell. Once the NStable code has been shown to adequately model these results, it can be applied to other problems.

Additionally, the NStable code could be coupled to a thermohydraulics code to fully simulate the reactor conditions during a transient. With this method, the incremental property changes at the end of each run could be taken directly from the coupled thermohydraulics code, which would use the neutron flux and power density

to calculate the these changes. If possible, the thermohydraulics code should communicate with NStable using MPI pipes, or even be added to the code itself as a C++ class. If communication between the codes must rely on file input and output, then the coupled simulation will be significantly slower.

6.4.2 Additional Run-Level Calculations

The NStable code could also be used to calculate the criticality of the simulation world using the k-eigenvalue method. The simplest way to implement this would be to calculate k_{eff} using the generational algorithm employed by MCNP. That is,

$$k_{eff} = \frac{\text{Number of neutrons produced through fission in one generation}}{\text{Number of neutrons produced through fission in the previous generation}} \tag{6.1}$$

where each run of the simulation only encompasses a single generation [4]. As noted previously, this method is only valid for near-critical systems, but having this option in NStable could be useful for simulating small perturbations of a such system. It would also be useful when comparing NStable to other reactor simulation codes that use this method to calculate criticality.

6.4.3 Isotopic Evolution

While the NStable code focuses on short time scales relative to traditional burnup calculations, it could be used to calculate the time dependent isotopic evolution of reactor materials. The most likely method for this calculation would be to run the NStable code for a short time period (T_{run}, approximately one second), during which the isotope production and losses are tallied for important isotopes; tallying every

isotope produced or lost would result in extremely complex reactor materials and would slow the simulation. After each short interval, the simulation would be moved forward in time by a large time step (T_{jump}, on the order of days), and the isotopic change at the start of the next simulation interval (short time period) could be calculated by prorating the tallies from the previous interval. That is

Algorithm 6.1: Isotopic evolution calculation

Input: Simulation with known material properties, decay chains for each isotope, reactor power level P

Output: Isotopic composition at a later time

while $t < t_{end}$ **do**

`/* Short simulation interval` $(t \rightarrow t + T_{run} = t_f)$ `*/`

Simulate neutron population in world

Tally isotope production (N_i^{prod}) and loss (N_i^{loss})

`/* Long time jump` $(t \rightarrow t_f + T_{jump})$ `*/`

Prorate isotope loss and production

$N_i^{prod}(t_f + T_{jump}) = A \, \frac{T_{jump}}{T_{run}} \, N_i^{prod}(t_f)$

$N_i^{loss}(t_f + T_{jump}) = A \, \frac{T_{jump}}{T_{run}} \, N_i^{loss}(t_f)$

Adjust tallies for decay

Scale tallies according to power level P

Change materials based on updated tallies

end

where A is some constant related to the change in the neutron flux between t_f and $t_f + T_{run}$. Note that the power level of the reactor will set the actual value of the neutron flux, which is on the order of 3×10^{14} neutrons per square centimetre per

second. Thus, the simulation results need to be scaled to a realistic flux level.

6.4.4 Nuclear Data Libraries

While the default G4NDL and MCNP-derived libraries are suitable for most situations, it would be useful to be able to create libraries in the G4NDL format directly from the ENDF/B data files. To accomplish this, the Python conversion scripts would need to be modified to parse the ENDF/B format and translate this to the G4NDL format. Alternatively, a nuclear data evaluation code, such as NJOY, could be used to create data libraries in the MCNP format from the ENDF/B evaluations [15]. Then these MCNP formatted libraries could be converted to the G4NDL format using the existing Python script. Additionally, NJOY can also Doppler broaden the cross sections to any required evaluation temperature.

Similarly, the ability to use multiple data libraries at different evaluation temperatures in Geant4 would reduce computation times. In this scheme, Geant4 would sample cross sections from the library with the closest evaluation temperature to the temperature of the material that the neutron is travelling through. This would reduce the amount of on-flight Doppler broadening necessary when sampling interaction cross sections. However, the NeutronHP datasets, and possibly the NeutronHP processes, would have to be redefined to implement multiple data library handling. This modification would also increase the initialization time of the simulation since multiple data libraries would need to be loaded.

References

[1] J. J. Duderstadt and L. J. Hamilton, *Nuclear Reactor Analysis*. John Wiley and Sons, Inc., 1976.

[2] K. O. Ott and R. J. Neuhold, *Introductory Nuclear Reactor Dynamics*. American Nuclear Society, 1985.

[3] S. Agostinelli *et al.*, "Geant4 - a simulation toolkit," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 506, no. 3, pp. 250 – 303, 2003.

[4] X-5 Monte Carlo Team, *MCNP - A General Monte Carlo N-Particle Transport Code, Version 5*. Los Alamos National Laboratory, 2005.

[5] J. Allison *et al.*, "Geant4 developments and applications," *Nuclear Science, IEEE Transactions on*, vol. 53, no. 1, pp. 270 –278, 2006.

[6] J. Apostolakis *et al.*, "Progress in hadronic physics modelling in geant4," in *Journal of Physics: Conference Series*, vol. 160, IOP Publishing, 2009.

[7] E. Mendoza, D. Cano-Ott, C. Guerrero, and R. Capote, "New evaluated cross section libraries for the GEANT4 code," tech. rep., International Atomic Energy Agency, 2012. INDC(NDS)-0612.

[8] J. Wellisch, "Geant4 physics validation for large hep detectors," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 502, no. 2âĂŞ3, pp. 669 – 672, 2003. Proceedings of the VIII International Workshop on Advanced Computing and Analysis Techniques in Physics Research.

[9] E. E. Lewis and J. W. F. Miller, *Computational Methods of Neutron Transport*. John Wiley and Sons, Inc., 1984.

[10] D. E. Cullen, C. J. Clouse, R. Procassini, and R. C. Little, "Static and dynamic criticality: Are they different," Tech. Rep. UCRL-TR-201506, Lawrence Livermore National Laboratory, 2003.

[11] M. Herman and A. Trkov, *ENDF-6 Formats Manual*. Brookhaven National Laboratory, rev. 1 ed., 2010. Report BNL-90365-2009.

[12] "Geant4." Computer Software, 2011. version 4.9.4.p02.

[13] "Geant4." Computer Software, 2012. version 4.9.5.p01.

[14] T-2 Nuclear Information Service, "Understanding NJOY," tech. rep., Los Alamos National Labs, 2000. LA-UR-00-1538.

[15] R. MacFarlane and A. Kahler, "Methods for processing ENDF/B-VII with NJOY," *Nuclear Data Sheets*, vol. 111, no. 12, pp. 2739 – 2890, 2010. Nuclear Reaction Data.

[16] GEANT4 Consortium, *Physics Reference Manual*, version 4.9.4 ed., 2010.

[17] G. Marleau, "Dragon theory manual: Part 1: Collision probability calculations," Technical Report IGE-236 Rev. 1, École Polytechnique de Montréal, 2001.

[18] G. Marleu, A. Hébert, and R. Roy, *A User Guide for DRAGON 3.06*. École Polytechnique de Montréal, rev. 1 ed., 2010. Report BNL-90365-2009.

[19] S. M. Ross, *Introduction to Probability Models*. Elsevier Inc., 9th ed., 2007.

[20] C. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656, 1948.

[21] C. Bungau, R. Barlow, and R. Cywinski, "Study on neutronics design of an accelerator driven subcritical reactor," in *IPAC'10 - Proceedings of the 1st International Particle Accelerator Conference*, pp. 160–162, Kyoto, Japan: IPAC 10 OC/ACFA, May 2010.

[22] M. R. Ball, A. C. Morreale, D. R. Novog, and J. C. Luxat, "The effect of supercell calculations due to modeling candu fuel pin clusters as annuli," in *Proceedings of the International Conference of Mathematics and Computational Methods Applied to Nuclear Science and Engineering*, MC 2011, (Rio de Janeiro, RJ, Brazil), American Nuclear Society, 2011.

[23] L. S. Waters, *MCNPX User's Manual, Version 2.3.0*. Los Alamos National Laboratory, 2002. LA-UR-02-2607.

[24] D. E. Cullen, "Tart 2005 a coupled neutron-photon 3-d, combinatorial geometry time dependent monte carlo transport code," tech. rep., Lawrence Livermore National Laboratory, 2005. UCRL-SM-218009.

[25] M. Shayesteh and M. Shahriari, "Calculation of time-dependent neutronic parameters using monte carlo method," *Annals of Nuclear Energy*, vol. 36, no. 7, pp. 901 – 909, 2009.

[26] L. Russell, A. Buijs, and G. Jonkmans, "A method for simulating real-time neutron populations, materials and geometries using the geant4 monte carlo toolkit," in *Proceedings of the 33rd Annual Conference of the Canadian Nuclear Society*, CNS 2012 SNC, (Saskatoon, SK, CA), Canadian Nuclear Society, 2012.

[27] R. Procassini, D. Cullen, G. Greenman, and C. Hagmann, "Verification and Validation of MERCURY: A Modern, Monte Carlo Particle Transport Code," in *Proc. of The Monte Carlo Method: Versatility Unbounded in a Dynamic Computing World*, pp. 17–21, American Nuclear Society, 2005.

[28] D. Cullen, R. Blomquist, C. Dean, D. Heinrichs, L. Kalugin, M. Lee, Y. Lee, R. MacFarlane, Y. Nagaya, and A. Trkov, "How accurately can we calculate thermal systems?," *University of California, Lawrence Livermore National Laboratory*, 2004. UCRL-TR-203892.

[29] R. Mosteller, "Validation suites for MCNP," in *Proceedings of the American Nuclear Society Radiation Protection and Shielding Division 12th Biennial Topical Meeting*, (Santa Fe, New Mexico), American Nuclear Society, 2002. LAUR-02-0878.

[30] R. Roy, G. Marleau, J. Tajmouati, and D. Rozon, "Modelling of candu reactivity control devices with the lattice code DRAGON," *Annals of Nuclear Energy*, vol. 21, no. 2, pp. 115 – 132, 1994.

[31] J. L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[32] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley Publishing Company, 3 ed., 2000.

[33] G. Cooperman, V. H. Nguyen, and I. Malioutov, "Parallelization of geant4 using top-c and marshalgen," in *Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, NCA '06, (Washington, DC, USA), pp. 48–55, IEEE Computer Society, 2006.

[34] K. Cronin, "Select a random N elements from list<T> in c#." Retrieved online, 2008. http://stackoverflow.com/questions/48087/select-a-random-n-elements-from-listt-in-c-sharp/48089#48089.

Appendix A

C++ Basics

Since Geant4 is written in object-oriented C++, a few of the core C++ concepts will be explained here [32].

A.1 Classes and Objects

In C++, users interact with classes of objects and functions. A C++ class is an *abstract* definition that can be used to create a *concrete* object. It may contain both data and functions that are defined inside the class definition; these functions and data are referred to as member functions and member data because they belong to the class. For example, an integer (*int*) is a simple class in C++ that contains four bytes of data that are used to represent a number. However, the *int* class cannot be interacted with until an integer object is declared (*int a = 4;*). A more complicated example is the string class, which contains an array of characters, but also has member functions, such as the length function, which returns the number of characters in the string.

A.2 Container Classes

While many classes perform complicated actions through their member functions, container classes are used simply to store and access data. The *vector* class is a good example of this. A vector in C++ is essentially a one-dimensional array of objects (such as integers) that can grow or shrink in length. This is beneficial in C++ since basic arrays must have a fixed size due to memory allocation requirements.

A.3 Inheritance

Inheritance is widely used in Geant4. Inheritance is the process of *deriving* a more specialized class from a base class. The derived class retains all the data and functionality of the base class, but additional member data and functions may be added. Additionally, base class functions may be overridden in the derived class. An example of inheritance would be creating a shape class, with basic properties such as a name and a generic perimeter function, and then deriving a square and a circle class from the shape class to provide specific implementations of the perimeter function.

A.4 Pointers

Pointers in C++ are objects that store the address of another object in memory. Pointers have a unique class that they point to, so an integer pointer cannot be made to point to a string object even though the memory addresses of a string and an integer are the same size. The one caveat is that a base class pointer can point to a derived class object, although in most cases, the base class pointer is not able to access

the attributes of the derived class (member data and functions). Additionally, the object that is being pointed to may be interacted with by *dereferencing* the pointer.

A.5 Polymorphism

Polymorphism is also widely used in Geant4. If a member function in a base class is declared to be *virtual*, and it is overridden in a derived class, then when a base class pointer to a derived class object calls the virtual function, it actually gets the derived class version of that function. For example, let class B have a virtual function *print* that prints "Base class", and let class D be derived from B. Now if class D overrides *print* so that it now prints "Derived class", then a base class pointer (class B pointer) pointing to a derived class object (class D object) will print "Derived class" if the pointer calls the print function. Polymorphism allows classes in Geant4 to be derived from general base classes and then used in source code functions that know nothing of the user-created derived class.

A.6 Arrays

Arrays are one of the most basic structures in C++. A group of objects of the same type (i.e. all of the same class) are created sequentially in memory (RAM) and can be accessed by an index. For example, to access the second element of an array of five integers, the syntax is *array_name[1]* because array indices start at zero. Thus a large number of objects can be stored with one variable and accessed through the indexing operator (i.e. the "[#]" syntax). The major drawback of C++ arrays is that the length of an array (the number of objects stored in the array) is fixed. To

change the length of an array, the original array must be copied and the data must be entered into a new array of a different size. This operation is extremely inefficient and costly if length of an array is being incremented by one multiple times.

A.7 C++ Vectors

Vectors in C++ are essentially arrays with added functionality that address some of the drawbacks of arrays. First the size of a vector can be changed dynamically without always copying the entire vector; if the length of a vector is incremented, the vector reserves space for more elements than it currently needs in anticipation of further increases in the vector length. A vector will also return the number of elements currently stored in it (not done in an array).

A.8 Doubly-Linked List

A doubly-linked list in C++ is implemented as a collection of nodes that are ordered from 1 to N, where N is the total number of nodes. Each node in the list contains an object, and two pointers, one to the next node and one to the previous node. Since the nodes are linked by pointers, they do not have to be stored contiguously in memory, but this means that they cannot be accessed through an indexing operator. To access node i in a list, the list must be traversed from node 1 to node i, or from node N to node i, so accessing elements is slower than in an array or vector. However, nodes may be easily removed from the middle of a list by linking the pointers of adjacent nodes around the node to be deleted, and then simply deleting the node.

Appendix B

Parallel Processing in Geant4

Monte Carlo simulations are generally parallelizable using two methods: run-level parallelism, where independent runs are performed in parallel, and event-level parallelism, where independent events are performed in parallel [33]. Run-level parallelism is simpler because a run is the highest level of a standard Geant4 simulation (a simulation of one run), whereas independent events run in parallel must be controlled by a master process that deals with the simulation at the run-level.

B.1 Run-Level Parallelism

Run-level parallelism is better known as batch processing, which consists of running multiple independent simulations in parallel. Each simulation is identical except for different input values, such as a different random seed, and all important results are saved to output files. The final results are obtained by combining the results of each simulation, and performing analysis on the amalgamated data. Thus, individual simulations are self-contained and do not need to communicate with any other process.

B.2 Event-Level Parallelism

Event-level processing uses a master-slave topology to process events in parallel. The master process is responsible for sending events to the slaves to be processed and for receiving the results of each event; no particle tracking occurs on the master process. The slaves are dependent on the master for instructions, and once finished simulating the current event, the slave waits for the next event to be passed from the master. Given that the slaves have significantly more (combined) computing power than the master, as much of the simulation as possible should be completed by the slaves. When the master is completing run actions at the beginning and end of a run, the slaves sit idle. While event-level parallelism is more inefficient than batch processing, it is necessary for parallel applications where the current run depends on the results of the previous runs. This is the case when simulating a neutron population over several runs.

Event-level parallelism is achieved in Geant4 using a program called TOP-C (Task Oriented Parallel C/C++), which was written by Gene Cooperman at Northeastern University [33]. TOP-C is a custom parallel library written using the MPI (Message Passing Interface) interface, which is used for parallel applications in both Linux and Windows. TOPC provides four main functions [33]

Generate Task Input Generates a single task (event) and passes it to a slave to process (on master only).

Do Task Process a given task and pass the result back to the master (on slave only).

Check Task Result Check the result of a task for errors (on master only).

TOPC Master Slave At this point, the master generates task inputs and waits for task results, and the slave waits for task inputs and calculates task results.

In TOP-C, the master and slave simulations are almost identical. Both must build simulation worlds, initialize physics lists and create a run manager. However, when each simulation reaches the *TOPC Master Slave* function, the master asserts control and the slaves wait for instructions. In the implementation of TOP-C in Geant4, the run manager must be over-ridden by a new run manager that contains the TOP-C functionality.

The MPI interface allows streams of serialized data to be communicated between different processes. Therefore, container classes such as hit collections cannot simply be passed between the master and the slaves. The container object must first be *marshalled* at one end of the MPI pipe, and then *unmarshalled* at the other end. When a C++ object is marshalled, the data contained in the object is converted into a single string of serialized data, and during unmarshalling, a new copy of the object is built from the serialized string [33]. Since this process is not trivial for even simple objects, Marshalgen, an automating marshalling script written by Gene Cooperman, can be used to automatically generated marshalling and unmarshalling functions for C++ objects [33]. Therefore, custom data classes can be created to easily transfer data to and from the slaves.

Appendix C

NStable Classes and Files

All of the classes and files created for the NStable code are listed below with a short description for each class/file. For convenience, the classes (and files) have been split into five categories, and within each category, the classes are listed alphabetically.

Table C.1: Control classes and driver files

Class/File (*Parent Class*)	**Description**
NSRunManager (*G4RunManager*)	The main run manager which performs the simulations, and coordinates the user actions.
NStable.cc	The main driver file for the NStable program.
NStable.icc	The parallel driver file, which adds TOP-C functionality to NStable.cc and replaces NSRunManager with ParNSRunManager.
ParNSRunManager (*NSRunManager*)	Adds parallel processing to NSRunManager. It divides the simulation between the master and the slaves, and coordinates all communication.

Table C.2: Simulation world classes

Class/File (*Parent Class*)	**Description**
BareSphereConstructor (*NSWorldConstructor*)	Builds the simulation world from a single sphere of a homogeneous material.
C6LatticeConstructor (*NSWorldConstructor*)	Builds the simulation world from a CANDU 6 lattice cell with periodic boundaries.
CubeConstructor (*NSWorldConstructor*)	Builds the simulation world from a homogeneous lattice cell with periodic boundaries.
NSWorld (*G4VUserDetectorConstruction*)	Manages the simulation world constructors. Chooses one to build the simulation world based on user input.
NSWorldConstructor	Virtual base class for simulation world constructors. Provides common functionality for the world constructors.

Table C.3: User action classes

Class/File (*Parent Class*)	**Description**
NeutronSD (*G4VSensitiveDetector*)	The main scoring class. It categorizes all hits and tallies neutron production/loss.
NSEventAction (*G4UserEventAction*)	Action that completes all beginning and end of event tasks. It loads the results of the event into an NSEventData object.
NeutronGenerator	Helper class for the primary generator action that creates a new primary neutron from each entry in the survivors list.
NSPrimaryGeneratorAction (*G4VUserPrimaryGeneratorAction*	Creates the primary neutrons for each event, and stores the delayed neutron and survivors lists. It also performs the population renormalization at the beginning of each run.
NSRunAction (*G4UserRunAction*)	Performs any beginning and end of run tasks. This includes all the run-level analysis such as calculating k_{eff}.

Table C.4: Physics list classes

Class/File (*Parent Class*)	**Description**
BoundaryStepLimiter (*G4StepLimiter*)	The periodic boundary condition process.
HPNeutronBuilder (*G4VNeutronBuilder*)	A helper class that adds the proper physics model and the nuclear data class (NeutronHPCSData) to each NSHadron process.
HPNeutronPhysicsList (*G4VUserPhysicsList*)	Defines all physics processes, models and nuclear data for the simulation.
NeutronHPCSData (*G4VCrossSectionDataSet*)	Loads and samples nuclear cross section data. This derived class redefines the Doppler broadening algorithm so that it does not automatically assume the cross sections were evaluated at 0 K.
NeutronProcessBuilder	A helper class that instantiates all of the NStable process (but not the base Geant4 processes).
NSHadronCaptureProcess (*G4HadronCaptureProcess*)	Derived capture process that can set η_λ^i from PrimaryNeutronInfo.
NSHadronElasticProcess (*G4HadronElasticProcess*)	Same as NSHadronCapture but for elastic interactions.
NSHadronFissionProcess (*G4HadronFissionProcess*)	Same as NSHadronCapture but for fission interactions.
NSHadronicProcess	Contains common function for NSHadron processes to set η_λ^i at the start of tracking.
NSNeutronInelasticProcess (*G4NeutronInelasticProcess*)	Same as NSHadronCapture but for inelastic interactions.
NSProcessManager	Helper class for the NeutronSD class. Used to get the η_λ^i values from each hadronic process.
NSStepLimiter (*G4StepLimiter*)	Step limiting process that stops the neutrons at the end of each run.

Table C.5: Utility classes

Class/File (*Parent Class*)	**Description**
LoggingAction	Helper class used to provide the output stream to any other class. If output is to a file, it opens the file.
MaterialEnumerator	Contains two enumerators that specify either a material (e.g. fuel, coolant) or a property (e.g. temperature, density).
NSInterpManager	Manages all of the interpolation vectors used for material/geometry changes in NStable.
NSInterpVector	Data class that stores a two-dimensional table of values (x and y). It can sample x-values in the range of the table using a specified interpolation scheme.
ParseInput	Parses the input file, and sets simulation parameters from input file (or default values if undefined in input file). It provides access to the input data for all other classes.

Table C.6: Container classes

Class/File (*Parent Class*)	**Description**
Containers.hh	Header file that contains all template classes derived from the C++ standard template library (STL).
NeutronData	Marshallable container class that contains the survivor/delayed neutron data: current time, lifetime, position, momentum, and the η_λ^i values.
NSEventData	Marshallable container class that contains all the results of an event: survivors, delayed neutrons, fission sites and event tallies.
NSPrimaryData	Marshallable container class that contains all of the information necessary to start an event: primaries, random number seed, event number, and any material/geometry changes.
NSTrackInfo	Contains η_λ^i values for secondary (copied) neutrons created by the BoundaryStepLimiter.
PrimaryNeutronInfo	Contains the information needed to correct the lifetime and η_λ^i values for primary neutrons (cannot be done in the NSPrimaryGeneratorAction). Corrections occur in the NSHadron processes and NeutronSD.
PropertyChange	Marshallable container class that contains the information for a geometry/material change in the simulation world (material, property, and new value).
TallyHit (*G4VHit*)	Contains all the tallies, survivors and delayed neutrons from the NeutronSD sensitive detector for a single event.
Triple	Template used to create classes that store three objects (of arbitrary classes).
TripleFloat	Marshallable class that is used instead of G4ThreeVector to define the position/momentum in NeutronData so that NeutronData can be marshaled.

Appendix D

NStable Target Selection Algorithm

The target selection algorithm (Algorithm 4.1) used in the neutron population renormalization process is designed to select m items from a sample size of N_s without repeating a selection. Moreover, this algorithm is designed to select the m items uniformly without any bias; that is, each of the items should have the same probability of being selected. This process is generally known as combing, and the problem of selecting m items from N_s uniformly has several solutions [10]. However, given the added constraint of selecting each item only once, a slightly different approach was needed. Algorithm D.1 was taken from an online coding forum, and therefore, it needed to be tested to verify that it did indeed select targets without bias [34].

Algorithm D.1: Target selection for deletion or duplication

Input: Number of primary neutrons N_p, number of survivors N_s

Output: List of targets for deletion or duplication

$m = |N_s - N_p|$ `// Number of missing/extra neutrons`

for $i = 0 \rightarrow (N_s - 1)$ **do**

  $r \in [0, 1]$ `// Generate a random number between 0 and 1`

  if $r < \frac{m}{N_s - i}$ **then**

    Add survivor i to list of targets

    $m = m - 1$

  end

end

The uniqueness criterion can be verified simply by examining Algorithm D.1. The algorithm passes once over each of the N_s items, and each item may be selected or not. Therefore, it is impossible for an item to be selected more than once. If $m > N_s$, then all N_s items would be selected, and the algorithm would restart with $m' = m - N_s$. As long as $m' > N_s$, all N_s items are selected, and then the selection algorithm restarts with m' decremented by N_s. In the NStable code, this situation could happen if the population shrinks by more than a factor of two over a run. However, the uniqueness condition is still upheld on the last iteration of Algorithm D.1.

To verify the lack of bias, a simple simulation was performed: 20 items were selected from a total of 40, and this was repeated one million times. Therefore, each item should be selected 500,000 times plus or minus an error term to account for the stochasticity of the simulation. Since this is a counting process (Poisson process), the

error can be given by $\pm\sqrt{1E6}$ or ± 1000 [19]. On a relative scale, the error is $\pm 0.2\%$. Figure D.1 shows the results of this simulation. All forty items have less than $\pm 0.2\%$ bias, and therefore, the selection algorithm is within the expected error. From this, it can be concluded that Algorithm D.1 is unbiased.

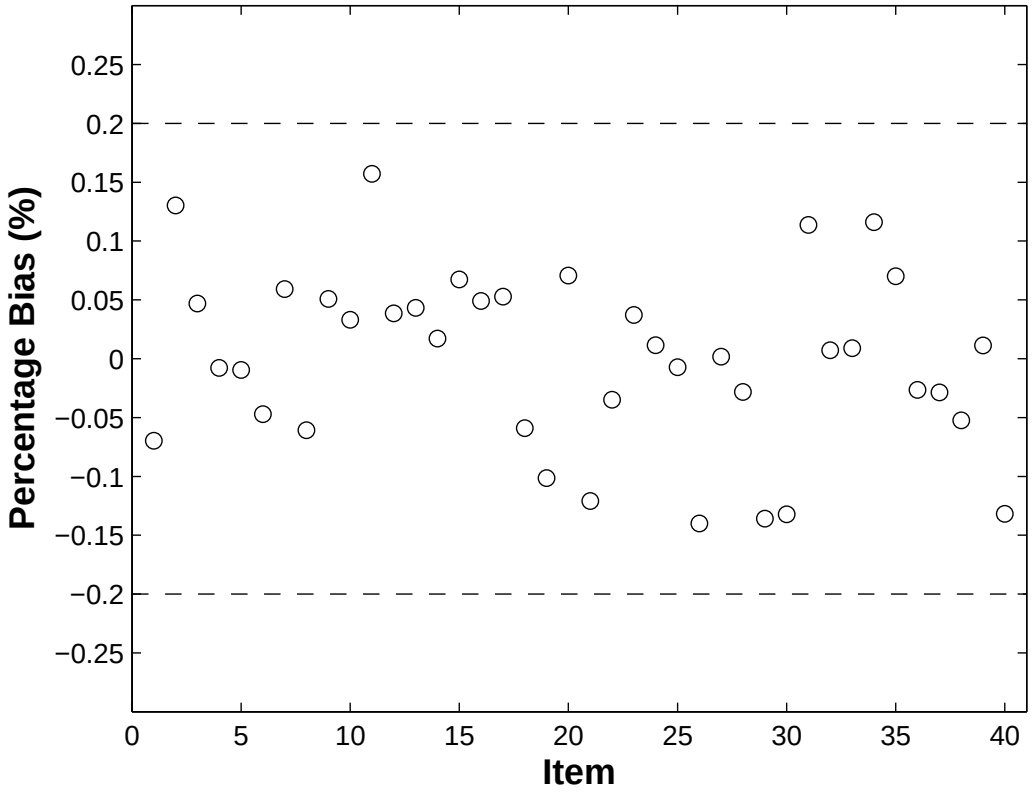


Figure D.1: Selection algorithm bias.

Appendix E

NStable Input Parameters

Table E.1: NStable Environment Variables

Environment Variables	
G4NEUTRONHPDATA	Location of the nuclear data for NeutronHP processes (cross sections and final state data).
PARMODE	Set to "MPI" for parallel processing and "SEQ" for sequential.
G4USE_TOPC	Set to "1" to use the TOP-C libraries either in sequential or parallel mode.

Table E.2: Simulation parameters and options

Required Parameters	
WORLD	Simulation world type (e.g. bare sphere or lattice cell)
NUM_RUNS	Number of runs in the simulation
NUM_EVENTS	Number of events simulated per run
NUM_PRIMARY_PER_EVENT	Number of primary particles initialized at the start of each
RUN_DURATION	Duration of a single run (T_{RUN})
NEUTRON_ENERGY	Initial energy of primary neutrons in first run
Geometric Parameters	
RADIUS	Sphere radius (sphere)
PITCH	Lattice pitch (lattice cells)
TEMP	Temperature of material (sphere and homogeneous lattice cell)
U235_CONCENTRATION	Concentration of U235 relative to U238 for NU material
HW_CONCENTRATION	Concentration of heavy water relative to NU in UHW material
Optional Parameters	
SEED	Initial random seed value
PERIODIC_BC	Enable periodic boundary conditions
INSTANT_DELAYED	Produce neutrons instantaneously
RENORMALIZE	Renormalize neutrons after each run
SE_MESH	Number of mesh steps in 3D for Shannon entropy calculation
Data File Options	
INITIAL_SOURCE_FILE	Load initial primaries and delayed neutrons from given source file
OUTPUT_LOG	Output overall results of each run to a log file
OUTPUT_SOURCE	Output survivor and delayed distribution to a file
OUTPUT_SRC_FREQ	Interval in runs when the source is saved to a file
FUEL_TEMP_DATA	Fuel temperature distribution with respect to time
COOLANT_TEMP_DATA	Coolant temperature distribution with respect to time
COOLANT_DENSITY_DATA	Coolant density distribution with respect to time

Appendix F

Example Files

F.1 NStable Files

F.1.1 Main Driver Files

Listing F.1. NStable driver file

```
/*
NStable.cc

Created by:  Liam Russell
Date:   22-06-2011
Modified:  09-07-2012

Main driver file for NStable code. Instantiates run manager, required classes
and optional classes. Also instantiates the parse input class (input
variables).
*/

// Include header files
//#define G4TIME

```

```
#include "G4Timer.hh"

#include "NSRunManager.hh"
//#include "G4UIQt.hh"
#include "G4VisExecutive.hh"
#include "G4UIExecutive.hh"
#include "G4UImanager.hh"
#include "G4UIsession.hh"
#include "G4UIterminal.hh"
#include "G4ThreeVector.hh"
#include "G4UnitsTable.hh"

#include "NSWorld.hh"
#include "HPNeutronPhysicsList.hh"
#include "NSPrimaryGeneratorAction.hh"
#include "NSRunAction.hh"
#include "NSEventAction.hh"
#include "LoggingAction.hh"

#include "Randomize.hh"
#include <fstream>
#include "ParseInput.hh"

#include "G4EventManager.hh"
#include "G4TrackingManager.hh"


// Include parallel processing file
#include "NStable.icc"


int main(int argc, char **argv)
{
   // Set up and start the timer
   G4Timer mainTimer;
   mainTimer.Start();
```

```
   // Simulation Variables --------------------------------------------------
55
   // Variables not set from input file
   LoggingAction *logSession = NULL;
   std::ostream *output = NULL;
   G4bool master = true;
60 G4bool readInput = false;
// G4long randomSeed;


#ifdef G4USE_TOPC
65 master = G4bool(TOPC_is_master());
#endif


   // Parse input file
70 ParseInput *InFile = new ParseInput;
   readInput = InFile->ReadInputFile(G4String(argv[1]));

   if(!readInput)
   {
75    G4cerr << "Unable to open " << G4String(argv[1]) << ". Exiting."
           << G4endl;
      return 1;
   }


80
   const ParseInput* infile = InFile;


   // Determine whether logging is required and set the output stream
85 if(infile->SaveResultsToFile() && master)
   {
// logSession = new LoggingAction(infile->GetResultsFile());
      logSession = new LoggingAction(infile);
      output = &(logSession->GetOutput());
```

```
90      }
        else
        {
           output = &G4cout;
        }
95

        // Random number generator --------------------------------------------------

        // Set up random engine
100     CLHEP::HepRandom::setTheEngine(new CLHEP::RanecuEngine);

        if(master)
        {
    //  randomSeed = infile->GetRandomSeed();
105

           CLHEP::HepRandom::setTheSeed(infile->GetRandomSeed());

           CLHEP::HepRandom::showEngineStatus();
           CLHEP::HepRandom::saveEngineStatus();
110     }


        // Simulation structures ----------------------------------------------------

115

        NSRunManager* runManager = new NSRunManager(infile, logSession);

        runManager->SetVerboseLevel(0);

120     // Set version string in the ParseInput object
        InFile->SetG4VersionString(runManager->GetVersionString());

        // Construct the world
        NSWorld *theWorld = new NSWorld(infile);
125     runManager->SetUserInitialization(theWorld);
```

```
    // Construct the physics list
    runManager->SetUserInitialization(new HPNeutronPhysicsList(infile));

    // Construct the primary generator
    NSPrimaryGeneratorAction * genAction;

    genAction = new NSPrimaryGeneratorAction(infile, master);

    runManager->SetUserAction(genAction);



    // Set optional user action class

    // Construct the run action only if this is the master
    NSRunAction *runAction = NULL;

    if(master)
    {
       runAction = new NSRunAction(genAction,logSession, infile);
    }

    runManager->SetUserAction(runAction);


    // Construct the event action
    NSEventAction* eventAction = new NSEventAction();
    runManager->SetUserAction(eventAction);


    // Start simulation ---------------------------------------------------------

    // Initialize the run
    runManager->Initialize();


    // Record the initialization time and restart the timer for the computation
```

```
   mainTimer.Stop();

   if(master)
   {
//  theWorld->DumpGeometricalTree();

      // Set world size in run action now that the world is built
      runAction->SetWorldDimensions(theWorld->GetWorldBoxDimensions());

      InFile->PrintInput(output);

      *output << "# Initialization time: " << mainTimer << G4endl
              << "#" << G4endl
              << "# ------------------------------------------------------------"
              << G4endl << "# " << G4endl
              << "# Starting Simulation" << G4endl
              << "#" << G4endl;
   }

   mainTimer.Start();

/*
#ifdef G4VISUALIZE
   // Visualization manager
   G4VisManager* visManager = new G4VisExecutive;
   visManager->Initialize();
#endif
*/

// G4EventManager *eventMan = G4EventManager::GetEventManager();
// G4TrackingManager *trackMan = eventMan->GetTrackingManager();
// trackMan->SetVerboseLevel(2);

   if(argc > 2)
   {
      G4UImanager* UImanager = G4UImanager::GetUIpointer();
      UImanager->ExecuteMacroFile(argv[2]);
```

```
    }
    // set to zero so function will go to terminal
    else if(infile->GetNumberOfEvents())
    {
205    runManager->BeamOn(infile->GetNumberOfEvents());
    }
    else
    {
       G4UIsession *ui = new G4UIterminal;
210
       ui->SessionStart();

       delete ui;
    }
215
    // Wrap up simulation --------------------------------------------------

    // Output the source convergence results
    if(master) runManager->OutputResults();
220
/*
#ifdef G4VISUALIZE
   delete visManager;
#endif
225 */

    // Stop the timer for the total computation time
    mainTimer.Stop();

230 if(master)
    {
       *output << "#" << "G4endl" << "# Total computation time: "
             << mainTimer << G4endl;
    }
235
    delete infile;
    delete runManager;
```

```
    delete logSession;

240 return 0;
}
```

Listing F.2. NStable TOPC driver file

```
/*
NStable.icc

Created by:  Liam Russell
 5 Date:   15-08-2011
Modified:  17-02-2011

This file is used to switch to parallel processing if the environment variable
G4USE_TOPC is set. Switches the run manager class from NSRunManager to
10 ParNSRunManager and overrides the main function.
*/


#ifdef G4USE_TOPC
15
// Include header files
#include "topc.h"
#include "ParNSRunManager.hh"

20 static int G4_main(int argc, char **argv);

int main(int argc, char **argv)
{
   // Set mode to no tracer messages
25 TOPC_OPT_trace = 0;

   TOPC_init(&argc, &argv);
   int ret_val = G4_main(argc, argv);
   TOPC_finalize();
30
```

```
    return ret_val;
}


#define NSRunManager ParNSRunManager
#define main G4_main

#endif
```

F.1.2 Input File

Listing F.3. NStable input file

```
## Script file for running Lattice Cell calculations

## World choice
WORLD  C6Lattice
PITCH 28.575

## World paramenters

## Run parameters
NUM_RUNS    150
NUM_EVENTS    6
NUM_PRIMARY_PER_EVENT   4200
RUN_DURATION    1e5
NEUTRON_ENERGY    1.0

## Options
SEED 1557136549
INSTANT_DELAYED 1

## Input source file
#INITIAL_SOURCE_FILE Src-C6Lattice.txt
```

```
## Output source/log files
#OUTPUT_LOG   Results/C6Lattice-Standard.txt
#OUTPUT_SOURCE   Results/Src-C6Lattice-Standard.txt
#OUTPUT_SRC_FREQ  1

## Run data for material changes
#COOLANT_TEMP_DATA   RunData/C6CoolantTempData.txt
#COOLANT_DENSITY_DATA   RunData/C6CoolantDensityData.txt
```

F.1.3 Output File

Listing F.4. NStable output file

```
# ------------------------------------------------------------
#
# Geant 4 Simulation of Neutron Stability
#
# Geant4 version Name: geant4-09-04-patch-02 (24-June-2011)
# Neutron Stability rev.60 (Bazaar build date 2012-07-29 13:31:29 -0400)
#
# Current time:  Wed Aug 8 21:40:37 2012
#
# Input Variables:
# World: Sphere
# Reactor Material: 92235
# Sphere radius (cm): 9.5
# Shannon entropy mesh: (20, 20, 20)
#
# Number of runs: 250
# Number of primaries per Event: 1667
# Number of events: 60
# Run duration (ns): 20
# Initial Neutron Energy (MeV): 1
#
# Data Library: /home/russellf/g4work/DataFiles/C6_ENDF6_T293.6
```

```
# Reactor temperature (K): 293.6
# Cross section temperature (K): 293.6
# Doppler Broadening Algorithm: Standard
# Random Seed: 2013092304
#
# Logging File: /usr2/scratch/russellf/U235Results/U235-MCENDF6-9.5cm-250runs-20.0ns
    -1.0MeV.txt
# Save source distribution interval: 250
# Output source file: /usr2/scratch/russellf/U235Results/Src-U235-MCENDF6-9.5cm-250runs
    -20.0ns-1.0MeV.txt
#
#
# Initialization time: User=23.7s Real=34.23s Sys=1.42s
#
# ------------------------------------------------------------
#
# Starting Simulation
#
# Number of Primaries per Run = 100020
# Number of Events per Run = 60
#
#
# Run # Start (ns) Lifetime (ns) Production krun keff Shannon H Duration (s)
#
    ------------------------------------------------------------------------------------------------------------

     1 0 4.099 786679 2.6560 1.266704 86.2654 18.89
     2 20 5.226 411441 1.3714 1.099257 88.6460 10.34
     3 40 5.443 410026 1.3731 1.100138 88.6941 10.12
     4 60 5.52 407959 1.3611 1.097124 88.7010 10.19
     5 80 5.536 412048 1.3824 1.102307 88.6907 10.12
     6 100 5.494 417409 1.3957 1.104748 88.7192 10.58
     7 120 5.528 408678 1.3722 1.100229 88.6531 10.27
     8 140 5.566 404285 1.3530 1.095686 88.6950 10.27
     9 160 5.514 414136 1.3879 1.103353 88.6703 10.21
    10 180 5.563 404423 1.3596 1.097609 88.6890 10.23
    11 200 5.515 414078 1.3887 1.103605 88.6863 10.1
```

```
 12 220 5.529 410065 1.3781 1.101606 88.6230 10.11
 13 240 5.503 417234 1.4027 1.106848 88.6785 10.48
 14 260 5.532 403667 1.3467 1.093972 88.7061 10.28
 15 280 5.575 406367 1.3649 1.098669 88.6564 10.03
 16 300 5.527 411184 1.3770 1.100971 88.6826 10.32
 17 320 5.573 403817 1.3491 1.094664 88.6815 10.33
 18 340 5.566 405252 1.3687 1.100116 88.6916 10.24
 19 360 5.548 405055 1.3507 1.094817 88.7254 10.21
 20 380 5.525 413187 1.3857 1.102970 88.7082 10.31
 21 400 5.58 398531 1.3290 1.090014 88.7137 10.32
 22 420 5.554 415816 1.3942 1.104759 88.6828 10.32
 23 440 5.551 404913 1.3617 1.098105 88.6694 10.38
 24 460 5.529 413815 1.3823 1.101800 88.7064 10.16
 25 480 5.561 406146 1.3682 1.099725 88.7107 10.13
.
.
.
 226 4500 5.553 409736 1.3676 1.098573 88.7215 10.66
 227 4520 5.576 400545 1.3443 1.094056 88.6944 10.71
 228 4540 5.58 408161 1.3764 1.101610 88.6768 10.51
 229 4560 5.547 409678 1.3728 1.100137 88.6743 10.07
 230 4580 5.585 399861 1.3361 1.091780 88.7312 10.27
 231 4600 5.589 405524 1.3667 1.099431 88.6938 10.28
 232 4620 5.546 412394 1.3878 1.103832 88.6530 10.55
 233 4640 5.515 412810 1.3822 1.102049 88.7139 10.59
 234 4660 5.543 409330 1.3741 1.100622 88.7116 10.42
 235 4680 5.549 409492 1.3754 1.100936 88.6727 10.01
 236 4700 5.545 407445 1.3674 1.099135 88.6735 10.37
 237 4720 5.535 411599 1.3756 1.100444 88.7309 10.36
 238 4740 5.531 410070 1.3656 1.097909 88.6813 10.17
 239 4760 5.556 409296 1.3745 1.100731 88.6867 10.47
 240 4780 5.526 411058 1.3826 1.102647 88.6974 10.58
 241 4800 5.528 410274 1.3696 1.099026 88.6977 10.47
 242 4820 5.555 406254 1.3600 1.097260 88.6999 10
 243 4840 5.508 413029 1.3895 1.104155 88.7129 10.66
 244 4860 5.558 408621 1.3671 1.098739 88.6652 10.61
 245 4880 5.523 409167 1.3742 1.100692 88.7195 10.09
```

```
  246 4900 5.573 405094 1.3544 1.095897 88.7316 10.15
  247 4920 5.546 408602 1.3762 1.101439 88.6970 10.48
  248 4940 5.58 400883 1.3438 1.093835 88.6872 10.06
  249 4960 5.551 408297 1.3670 1.098795 88.6926 10.34
  250 4980 5.517 409329 1.3718 1.099918 88.6550 10.47

# Total/Avg ( 216 runs): 5.545 408940 1.37 1.099409 88.6896 10

# Source convergence limit = 0.1%
# Source converged after 10 runs.
#G4endl# Total computation time: User=2.1e+02s Real=2.6e+03s Sys=14s
```

F.2 MCNP Input Files

Listing F.5. MCNP input for U235 sphere

```
Criticality of bare U235 sphere
C CELL CARDS
C Target sphere
1 1 -18.75 -11 imp:n=1
C Outside world (void)
99 0 11 imp:n=0

C SURFACE CARDS
C Sphere
11 so 8.7
C

C DATA CARDS
C Criticality control cards
kcode 5000 1.0 25 250
sdef erg=2.0 pos=0 0 0
C
C Material cards
```

```
m1 92235.66c -18.75
C
C Problem mode (neutrons)
mode n
C
C Cutoff card (turn off implicit capture)
C cut:n j j 0 j j
C
C
C Physics fission card (turn off total nu)
totnu no
C
```

Listing F.6. MCNP input for UHW sphere

```
Comparison between MCNP and GEANT4 - Neutrons impinging on a slab of material
C CELL CARDS
C Target sphere
1 1 -3.01482 -11 imp:n=1
C Outside world (void)
99 0 11 imp:n=0
C


C SURFACE CARDS
C Sphere
11 so 87.5
C


C DATA CARDS
C Criticality control cards
kcode 5000 1.0 25 250
sdef erg=2.0 pos=0 0 0
C
C Material cards
m1 92235.66c -0.000773
    92238.66c -0.106527
    8016.66c -0.713116
```

```
      1002.66c -0.179683
C mt1 hwtr.10t
C
C Problem mode (neutrons)
mode n
C
C Physics fission card (turn off total nu)
totnu no
```

F.3 DRAGON Input File

Listing F.7. DRAGON input for standard C6 lattice cell

```
*----
* 37 Element CANDU Bundle
* Based on example TCWu05
* Used for testing self-shielding options in DRAGON 3.06*
*----
* Define STRUCTURES and MODULES used
*----
LINKED_LIST
 LIBRARY CANDU6S CANDU6F VOLMATS VOLMATF PIJ FLUX BURNUP EDITION ;
SEQ_BINARY
 INTLINS INTLINF ;
SEQ_ASCII
 psmixs psmixf fluxes ;
MODULE
 LIB: GEO: NXT: PSP: SHI: ASM: FLU: EDI: END: ;
*----
* Microscopic cross sections from file endfb7gx format WIMSD4
*----
LIBRARY := LIB: ::
 NMIX 20 CTRA WIMS
 MIXS LIB: WIMSD4 FIL: iaea
```

```
MIX 1 561.285 0.8074
  Onat = '6016' 7.9986E+01
  D2D2O = '3002' 1.9889E+01
  H1H2O = '3001' 1.2455E-01
MIX 2 561.285 6.5041
  Nb93 = '93' 2.5800E+00
  Fenat = '2056' 4.6780E-02
  Crnat = '52' 8.0880E-03
  Ninat = '58' 3.5000E-03
  B10 = '1010' 2.4310E-05
  Zrnat = '91' 9.7313E+01
MIX 3 448.72 0.0012
  Cnat = '2012' 2.7110E+01
  Onat = '6016' 7.2890E+01
MIX 4 336.16 6.4003
  Fenat = '2056' 1.3500E-01
  Ninat = '58' 5.5000E-02
  Crnat = '52' 1.0000E-01
  Zrnat = '91' 9.8209E+01
  B10 = '1010' 5.9620E-05
MIX 5 336.16 1.08875
  O16 = '6016' 7.9893E+01
  D2D2O = '3002' 2.0098E+01
  H1H2O = '3001' 8.2535E-03
MIX 6 859.99 10.5541
  O16 = '6016' 1.18502E+1 1
  U235 = '2235' 6.26700E-1 1
  U238 = '8238' 8.75183E+1 1
  U234 = '234' 4.8E-3 1
MIX 7 COMB 6 1.0
MIX 8 COMB 6 1.0
MIX 9 COMB 6 1.0
MIX 10 561.285 6.3918
  Zrnat = '91' 9.8182E+01 1
  Fenat = '2056' 2.1000E-01 1
  Crnat = '52' 1.0000E-01 1
  Ninat = '58' 7.0000E-03 1
```

```
      B10 = '1010' 5.9620E-05 1
  ;
*----
* Geometry CANDU6S : cluster for self-shielding
* CANDU6F : annular cluster for transport
*----
CANDU6S := GEO: :: CARCEL 12 1 3
 X- REFL X+ REFL Y- REFL Y+ REFL
 MESHX -14.2875 14.2875
 MESHY -14.2875 -0.1 0.1 14.2875
 RADIUS 0.00000 0.74425 2.18350 3.60300 5.16890 5.60320 6.44780
        6.58750 8.07000 9.55250 11.03500 12.51750 14.00000
 MIX 1 1 1 1 2 3 4 5 5 5 5 5 5 1 1 1 1 2 3 4 5 5 5 5 5 5
     1 1 1 1 2 3 4 5 5 5 5 5 5
 CLUSTER ROD1 ROD2 ROD3 ROD4
 ::: ROD1 := GEO: TUBE 2 3 2
   MIX 6 10 6 10 6 10 6 10 6 10 6 10
   NPIN 1 RPIN 0.0000 APIN 0.0000
   MESHX -0.6540 -0.1 0.1 0.6540
   MESHY -0.6540 0.0 0.6540
   RADIUS 0.00000 0.6122 0.6540 ;
 ::: ROD2 := GEO: ROD1 MIX 7 10 7 10 7 10 7 10 7 10 7 10
   NPIN 6 RPIN 1.4885 APIN 0.0000 ;
 ::: ROD3 := GEO: ROD1 MIX 8 10 8 10 8 10 8 10 8 10 8 10
   NPIN 12 RPIN 2.8755 APIN 0.261799 ;
 ::: ROD4 := GEO: ROD1 MIX 9 10 9 10 9 10 9 10 9 10 9 10
   NPIN 18 RPIN 4.3305 APIN 0.0 ;
 ;
CANDU6F := GEO: CANDU6S :: SPLITR 1 16 16 16 3 5 3 4 4 4 4 4
 ::: ROD1 := GEO: ROD1 SPLITR 4 1 ;
 ::: ROD2 := GEO: ROD2 SPLITR 4 1 ;
 ::: ROD3 := GEO: ROD3 SPLITR 4 1 ;
 ::: ROD4 := GEO: ROD4 SPLITR 4 1 ;
 ;
*----
* Self-Shielding calculation EXCEL
* Transport calculation EXCEL
```

```
* Flux calculation for keff
*----
VOLMATS INTLINS := NXT: CANDU6S ::
 TITLE 'CANDU-6 CELL'
 EDIT 0 MAXR 300 TRAK TISO 8 20.0 SYMM 12 ;
psmixs := PSP: VOLMATS :: FILL RGB TYPE REGION ;
LIBRARY := SHI: LIBRARY VOLMATS INTLINS ::
 EDIT 0 ;
VOLMATF INTLINF := NXT: CANDU6F ::
 TITLE 'CANDU-6 CELL'
 EDIT 0 MAXR 300 TRAK TISO 14 20.0 SYMM 12 ;
psmixf := PSP: VOLMATF :: FILL RGB TYPE REGION ;
PIJ := ASM: LIBRARY VOLMATF INTLINF :: ;
FLUX := FLU: PIJ LIBRARY VOLMATF ::
 TYPE K EDIT 1 ;
EDITION := EDI: FLUX LIBRARY VOLMATF ::
 MERG REGION 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   1 2 3 4 5
   0 0 0 0 0 0 0 0 0 0
   12 11 10 9 8
   0 0 0 0 0 0 0 0 0 0
   13 14 15 16 17
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   40 39 38 37 36
   0 0 0 0 0 0 0 0 0 0
   41 42 43 44 45
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0
   6 7 0
   18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
   46 47
   48 49 50
   51 52 53 54 55
```

```
      56 57 58
      59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0
   STAT FLUX
   EDIT 3
   SAVE ON 'INITIAL' ;
 fluxes := EDITION ;
 END: ;
QUIT "LIST" .
```