MCMASTER UNIVERSITY

MASTER'S THESIS

Simulation of Subcritical Experiments in ZED-2 using G4-STORK

Author:
Salma Mahzooni

Supervisor:
Dr. Adriaan Buijs

A thesis submitted in fulfilment of the requirements
for the degree of Master's of Applied Science

in the

Department of Engineering Physics

March 2015

Abstract

In nuclear reactors, transients may happen, for example at the time of refueling, rod withdrawals or insertions, and during reactor accidents. Transient behavior may result from changes either in materials or in the geometry of the reactor core components. An enhanced understanding of these time-dependent changes in reactors may improve the reactor operation and reduce the probability of accidents. In spite of the importance of understanding the conditions that lead to transients, there are not many time-dependent reactor simulation codes available.

This study is focused on modeling the sub-critical reactivity measurements in the ZED-2 reactor, using the recently developed G4STORK computer code. The ZED-2 experiment measures the sub-critical state resulting from a step-wise reduction of the moderator level.

G4-STORK is a time-dependent Monte Carlo code for reactor neutronics calculations based on the GEANT4 toolkit. G4-STORK has the ability to follow the evolution of the neutron population in time, including delayed neutrons, and to model the resulting changes in material and geometric properties of a reactor.

The k_{eff} values calculated by G4-STORK were compared with the experimental measurements and with MCNP results. The comparison shows significant discrepancies with both MCNP and the experimental measurements. These discrepancies are increasing as the reactor becomes increasingly subcritical (from $\sim$20 mk to $\sim$40 mk). The recently developed G4-STORK code is still at an early stage of its development, and needs to be improved further to be used for transient analyses of the reactors. Given the flexibility of G4-STORK, there are many opportunity to improve and extend this code.

Acknowledgements

I would like to thank my supervisor, Dr. Adriaan Buijs for all his help and support for the past few years. I would also like to thank everyone in the nuclear engineering department at McMaster. In particular, I want to acknowledge folks in Dr. Buijs' group: Jason Sharpe, Wesley Ford, and Andrew Tan. You guys provided a great deal of help and welcome distractions during all the coding, debugging, and writing frustrations.

Contents

List of Figures

List of Tables

I dedicate my dissertation to my family and many friends. A special feeling of gratitude to my loving parents, Mehdi Mahzooni and Zahra Naghavi, whose words of encouragement and push for tenacity ring in my ears, who held my hands and had my back through all difficulties that I have been through. My sisters Samaneh and Sarah and my brothers Samer and Hamid have never left my side and are very special.

Chapter 1

Introduction

Nuclear power plants perform tasks similar to other generating power plants (such as oil, gas, etc). The heat produced from the fission reactions is used to generate steam which causes the turbines to run and generate electricity. Although the concept of producing power is similar to other conventional power generating plants, the nuclear reactor core is an intricate system. The study of the nuclear reactor entails a study of the combination of the neutron transport theory and heat transfer.

It is hard to observe what exactly happens inside an operating reactor. Studying and understanding the behavior of an operating nuclear reactor therefore relies heavily on computer simulations. Computer simulations and modeling are alternatives to experimentation, and cheaper. Using computer simulations allows scientists and engineers to model and observe complex problems and situations such as dynamic states and accident cases. There are two methods used to simulate and model nuclear reactors: the deterministic and Monte Carlo methods.

The deterministic method is based on solving the neutron transport equation directly while making some assumptions (e.g. discretizing energy and space) to simplify this equation. Deterministic neutron transport codes are widely used to simulate the physics of reactors.

The Monte Carlo method simulates the life of single neutrons from birth to death in continuous-energy and three-dimensional geometry. It is truer to nature, but more time-consuming than deterministic codes.

The focus of this study is on dynamic neutron transport theory using the Monte Carlo method.

1.1 Scope of the study

This study has been built around modeling a subcritical experiment in the ZED-2 reactor in a Monte Carlo reactor physics simulation code, the so-called G4-STORK code. This code was developed at McMaster University. The title of this code has been given the name "Geant4 STOchastic Reactor Kinetics" or G4-STORK.

This study is based on subcritical measurements done with ZED-2 in the Canadian Nuclear Laboratories, (CNL, formerly known as AECL) at Chalk-River, Ontario [4]. The investigation was conducted to demonstrate the ability of the G4-STORK code for reactor kinetics calculations. The aspects looked into were how close the simulation results are to the experimental measurements; also the G4-STORK results were compared to another Monte Carlo reactor physics code known as Monte Carlo N-Particle (MCNP) code, although the MCNP reactor code is a static simulation code.

1.1.1 Status of the G4-STORK code

The development of G4-STORK started in September 2010 as a Master's project [6]. Code capabilities and preliminary results were introduced in two reactor physics conferences (CNS) in 2011 and 2012 as well as in the dissertation [7] [6]. The G4-STORK code can be classified as a three-dimensional, continuous-energy, time-dependent, Monte Carlo code. This code is based on the Geant4 tool kit and is intended only for reactor

physics calculations. It must be pointed out that this code has been only tested and used by the McMaster university group, and this thesis is not a full description of G4-STORK. Only the essential calculation methods and code capabilities are fully covered in this text (mainly the features that were required to model the aforementioned subcritical experiment in the ZED-2 reactor). New features are constantly being developed, implemented, and envisioned for G4-STORK, features which are the basis of the upcoming thesis projects.

1.1.2 Structure of the Thesis

This thesis is divided into two parts. Part I: the theoretical background to the fundamental topics of this study, Part II: the scientific value of this project, and the features and capabilities of the G4-STORK code which are essential to model the experiment.

Chapter 2 provides the basics of the theoretical physics background essential to this project. In this chapter, an introduction to the physics of neutron interactions is given, followed by the derivation of the neutron transport equation and its simplified version, the diffusion theory. This chapter includes information on both the statics and kinetics of the neutron transport equation. Chapter 3 is about the methods that are used to compute the solutions to the neutron transport equations. These methods are then separately discussed and defined in the detail necessary for this project. At the end of this chapter, a brief comparison of the advantages and disadvantages of these two methods is provided. This theoretical background, in terms of both physics and computational, is collected from various different sources [2][1][3]

The second part of this thesis starts with chapter 4, which is a brief discussion of the differences and similarities of this study with respect to other studies that have been published in the literature. G4-STORK methodology is then defined in chapter 5. This chapter is divided into two parts: G4-STORK mechanisms and the discussion of the ZED-2 experiment, followed by the implementation of this experiment in G4-STORK.

Also, the implementation of this very experiment in MCNP is very briefly defined in this chapter. Chapter 6 presents the data and discusses the results in detail. The final chapter is left for conclusions and some future plans to further developments in G4-STORK.

1.1.3 Scientific Value of This Work

The unique feature of this study is that it uses a new transport calculation code that has been recently developed. The G4-STORK code is based on an open-source toolkit which makes this code much more user-friendly than any other existing reactor Monte Carlo codes. This code uses the data from a recent release of the evaluated data libraries (i.e. ENDF/B-VII, etc), data which then are converted to the GEANT4 format.

The characteristic feature of G4-STORK is that it is only used in reactor physics calculations, specifically for time-dependent calculations. The time-dependency feature of the G4-STORK code makes this code very special as it can estimate the kinetic neutrons' behavior in a nuclear reactor. This feature of the G4-STORK code makes it a good candidate for this study.

1.1.3.1 Motivations

The neutron population in a nuclear reactor fluctuates over time even when the core is in a stable (critical) state. These fluctuations in neutron population are then periodically suppressed by the reactor's control system to sustain a stable neutron population throughout the nuclear reactor operation. However, the time-dependent response of the neutron population becomes important when the state of the nuclear reactor changes with time (e.g. at a time of accident or a power maneuver). Thus, the time-dependent behavior of the neutron population in a nuclear reactor is one of the most important features of nuclear reactor operation.

The evolution of the state of a nuclear reactor has been studied using mostly deterministic approaches to the time-dependent neutron transport equation such as the point-kinetics approximation. These approximations suffice to describe the mean value of the state variables in a deterministic manner. Since these approximations are deterministic, they are limited by the same static deterministic limitations (discretizations of space, time, and energy) that are important factors in solving the time-dependent (kinetics) behavior of the reactor. Nonetheless, this kinetics behavior of a nuclear reactor is a stochastic process and can be defined in sets of stochastic kinetic equations

The stochastic (Monte Carlo) techniques do not rely on such limitations. In Monte Carlo methods, all the neutrons can be tracked both in time and space in a nuclear reactor core. Thus, the overall properties of a nuclear reactor can be simulated at any time during the simulation. Many Monte Carlo reactor physics codes, such as MCNP, neglect the time dependency for calculating the reactor characteristics. Therefore, these calculations can only accurately simulate the static state of a reactor and not the kinetic state of a reactor. As stated above, G4-STORK has the capability to simulate both the reactor response to transient behavior and time-dependent systems.

G4-STORK tracks the evolution of the reactor's neutron population over time and determines various kinetic parameters. From the simulated parameters the k_{eff} of the system in time can be extracted. This study is to model the subcritical ZED-2 experiment to compute k_{eff} at various moderator heights for the subcritical state. However, this study goes beyond merely deducing the results of this experiment using G4-STORK. The following summarizes the goals, objectives, and questions of this study.

Study Goals, Objectives, and Research Questions

- **Study Goals:**
  - Demonstrate the capabilities of the G4-STORK code;
  - Develop an understanding of the modeling reactor physics experiment using the G4-STORK code;
- **Objectives:**
  - Model an experiment using G4-STORK to test and explore the performance of this reactor physics computer code for simulations of nuclear reactor kinetics;
  - Revise and refine the preliminary conceptual modeling in G4-STORK to model the real reactor physics experiment for G4-STORK development;
- **Research Questions:**
  - How do the G4-STORK results compare to experimental and MCNP results?
  - Which are the important characteristics of this experimental model leading to further improvement of this code for better reactor kinetics calculations in future?
  - What working features are warranted based on G4-STORK development to guide future research?

Part I

Theoretical Background

Chapter 2

Theory

2.1 Physical Background

In the first few sections of this chapter, the basic physics of neutron interactions and the necessary principles of nuclear reactors are discussed. Understanding of the physical background is important in Monte Carlo simulations, where every neutron is tracked through its interaction with existing material and boundary conditions. Also, these fundamental concepts help in understanding neutron transport theory. The latter sections are devoted to introducing the general form of the time-dependent neutron transport equation.

The purpose of this discussion is to familiarize the reader with fundamental neutron transport theory. Readers who desire to see a more detailed discussion on the subject may wish to consider one of the many related textbooks [2][1][3].

2.1.1 Neutron Interactions and Cross Sections

The primary goal of the reactor physics is to understand how neutrons are distributed in space, time, and energy in a nuclear reactor [2]. The neutron transport equation satisfies

this goal by describing the motion of neutrons and their interactions with the materials in a nuclear reactor.

Neutrons interact with any nuclei at any energy and temperature in the nuclear reactor materials. As stated above, the neutron transport theory depends on understanding of the neutrons interactions throughout their journey within the nuclear reactor before they are absorbed or leaked. The transport equation is a linear equation since the neutron-neutron interactions are negligible in such systems which have much higher atomic density than neutrons density in a given medium [8].

The probability that one of the neutron interactions occurs between the target nucleus and an incident neutron is called microscopic cross section [9]. The cross sections concept is more described in the next section 2.1.1.2.

2.1.1.1 Cross Sections

The concept of cross section can be understood by considering a neutron passing through a thin layer of a material with an area A[1] that has N' nuclei as shown in Figure 2.1. The cross sectional area of each nucleus is set to be σ. The probability which the neutron may have an interaction with one of these nuclei is $\frac{N'}{A}\sigma$. [2]

At the atomic level σ does not simply represent the geometric cross sectional area of the target nucleus, but rather it represents a physical quantity that depends on the specific target nucleus and energy of the incident neutron. However, the dimensions of this quantity remain as the dimensions of area, and usually it is given in units of barns (1 barn$= 10^{-24}$cm^2).

Therefore, the probability of having a certain type of interaction differs depending on the energy of the incident neutron. The macroscopic cross section for interaction type *i*

[1] A is the area of the slab $A = X \times Y$

[2] Where $\frac{N'}{A}$ represents the area density.

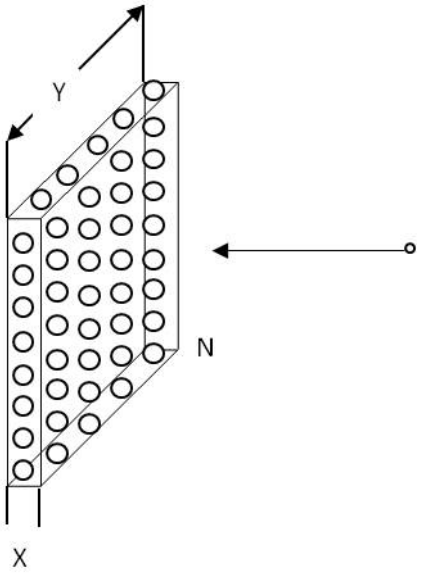


FIGURE 2.1: A neutron incident normally upon a thin layer(adapted from [1]).

is given by 2.1

$$\Sigma_i(E) = N\sigma_i(E) \tag{2.1}$$

where N is the atomic number density of the material, and i is one of the reactions as it was shown in figure 2.2.

The following subsections introduce the significant neutrons' interaction with nuclei in the nuclear reactors.

2.1.1.2 Neutron Interactions

Neutrons interact with matter in two general types: scattering and absorption [1]. The neutron interaction in which the neutron bounces off the nucleus with or without changing its energy is called scattering. The scattered neutron changes its direction, but the number of protons and neutrons in the nucleus stays unchanged. Scattering is classified into two categories: elastic and inelastic. When a neutron is absorbed by a nucleus, it can lead to either emitting various types of radiation or inducing fissions [9]. A neutron

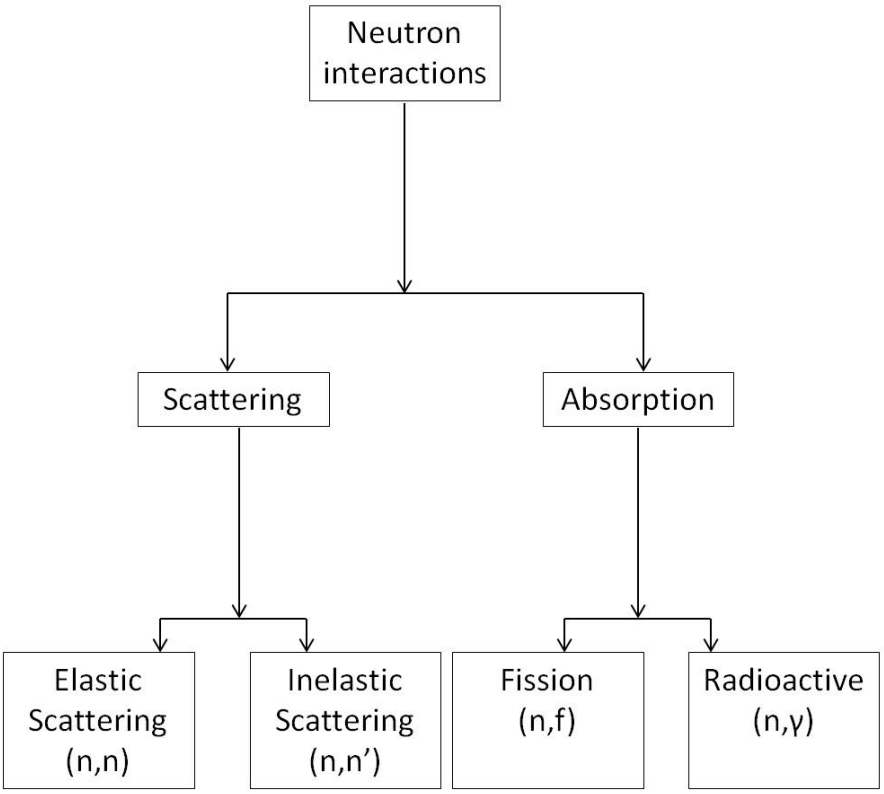


FIGURE 2.2: Primary neutron interaction types.

is absorbed mainly by radiative capture and fission interactions. Figure 2.2 shows the types of neutron interactions[3].

In **elastic scattering**(n,n), the incident neutron may be absorbed by the nucleus which results in the formation of the compound nucleus. However, the compound nucleus then re-emits the neutron and the compound nucleus returns back to the original nucleus. The total kinetic energy and momentum of the neutron-nucleus system are conserved [9]. Figure 2.3 shows elastic cross section plots of $^{238}_{92}$U and $^{235}_{92}$U. The resonant behavior of the nucleus cross section is observed in the elastic cross section plot. The resonance peaks are representative of the formation of excited states of the compound nuclei [2]. It is referred to as **potential scattering**, when a neutron is incident on a nucleus without any absorption (i.e. formation of a compound nucleus). In most scattering collisions,

[3]The notation (n,j) used in the parenthesis in Figure 2.2 represents the interaction of a neutron n with target nucleus that results in a resultant nucleus and an outgoing particle j.

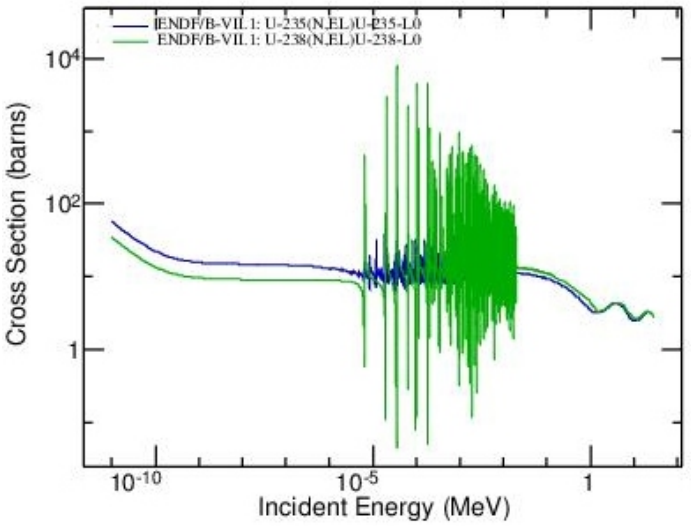


FIGURE 2.3: The elastic cross section of $^{238}_{92}$U and $^{235}_{92}$U (ENDF/B-VI, 300 K data).

the neutron penetrates into the nucleus and forms an excited state of the nucleus from which radiation is eventually released. This scattering interaction is called **inelastic scattering**(n,n$'$). In inelastic scattering, the total final kinetic energy is less than the initial total kinetic energy of the system, since a part of the initial energy was spent on exciting the nucleus. Some inelastic cross sections are plotted in Figure 2.4.

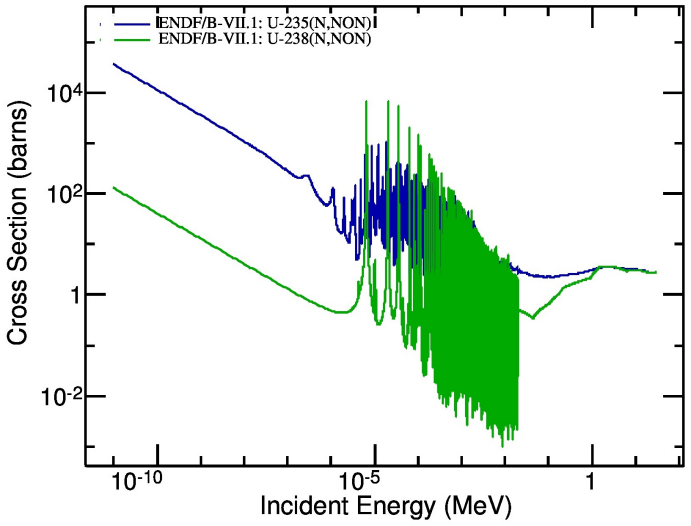


FIGURE 2.4: The inelastic cross section of $^{238}_{92}$U and $^{235}_{92}$U (ENDF/B-VI, 300 K data).

In the case of **radiative capture** (n,γ), the neutron is absorbed by the target nucleus, and the compound nucleus emits one or more high-energy photons. This interaction is

especially important in reactor physics, since it involves removing neutrons from the chain reaction [1]. The capture cross section of $^{238}_{92}$U is plotted in Figure 2.5. When the center-of-mass energy[4] E_c plus the binding energy of the incident neutron E_b is close to a nuclear energy level of the compound nucleus, the probability of forming the compound nucleus is large [1]. High probabilities of formation of the compound nucleus are responsible for the resonance behavior in capture cross sections at those energies.

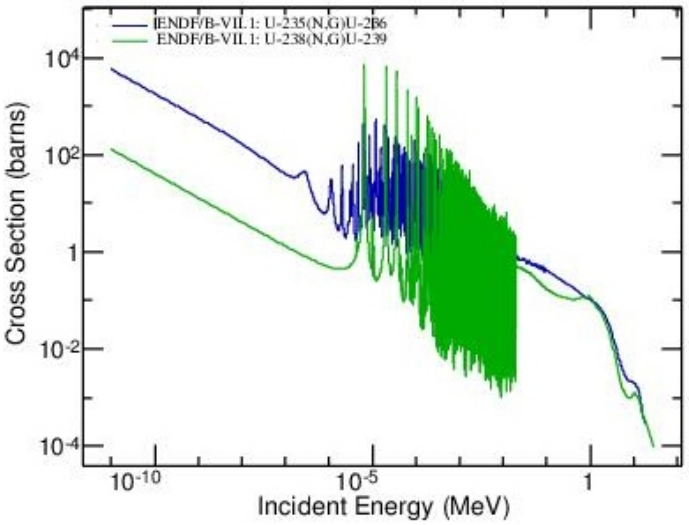


FIGURE 2.5: The radiative capture cross section of $^{238}_{92}$U and $^{235}_{92}$U (ENDF/B-VI, 300 K data).

Fission interactions (n,f) occur when an incoming neutron is absorbed by a nucleus and the nucleus splits into two lighter nuclei while releasing some (typically two or more neutrons). These neutrons, which are produced at the instant the fission reaction happens, are called prompt neutrons. The fission products' nuclei are highly unstable and will release neutrons later in the process. These neutrons produced from fission precursors are called delayed neutrons. Also, the similar resonance behavior of the cross sections for nuclear fission is expected since the mode of disintegration is relatively independent of the formation mechanism. The fission cross sections of $^{238}_{92}$U and $^{235}_{92}$U are plotted in Figure 2.6. The fission reaction occurs only for actinides, and for nuclei

[4] $E_c = (\frac{M}{m+M})E$ where M: nuclear mass, m: neutron mass, and E is the neutron energy in laboratory system

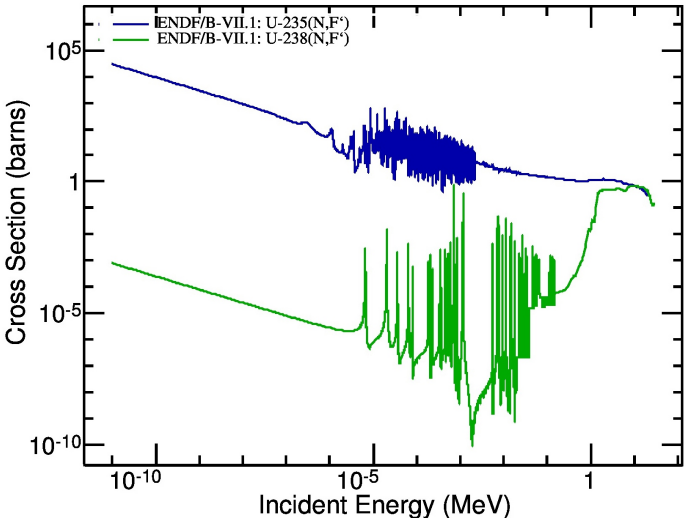


FIGURE 2.6: The fission cross section of $^{238}_{92}$U and $^{235}_{92}$U (ENDF/B-VI, 300 K data).

that have high energy neutrons (6-9 MeV) impinging on them that are able to overcome the binding energy of the nucleons in the nucleus [1].

2.1.1.3 Neutron Transport Equation

The nuclear fission reactors are based on the self-sustaining chain reactions. In every neutron-induced fission reaction, two or three neutrons are released so that the possibility of a sustained neutron chain reaction is obvious [2]. Figure 2.7 illustrates an induced-fission chain reaction.

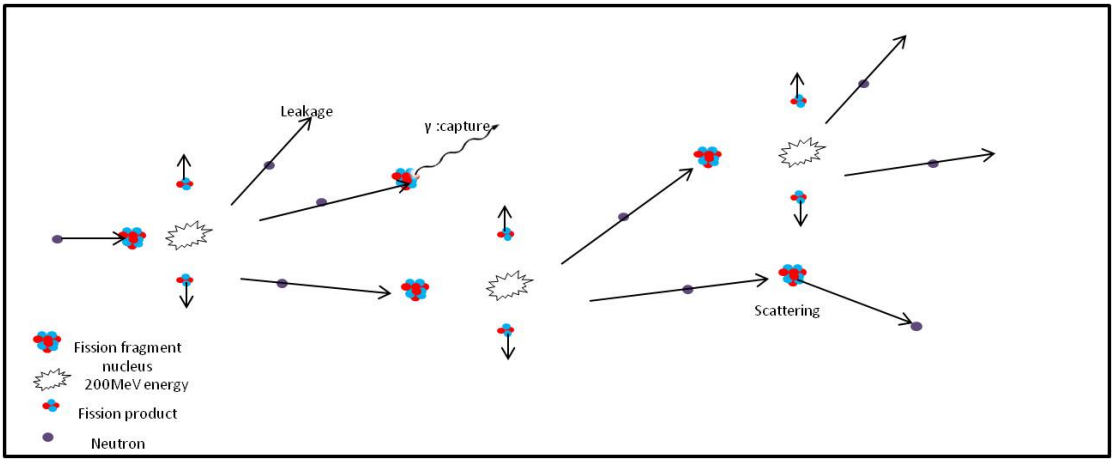


FIGURE 2.7: Neutron fission chain reaction (adapted from [2] chapter 2.)

Thus, to sustain a fission chain reaction at least one of the produced neutrons must survive to induce further chain reactions [2].

The neutron transport equation also known as the neutron Boltzmann equation describes the rate of nuclear reactions in a given boundary conditions. This equation is essentially an equation for the neutron conservation in a medium [10].

The most fundamental goal of a simulation code is to find the best estimate of the neutron loss and production rates. Therefore, the neutron transport equation is the most fundamental reactor physics equation solving the behavior of the neutron population within certain material using a given set of boundary conditions.

Figure 2.8 illustrates neutrons moving in the direction of solid angle $\mathrm{d}\Omega$ in an infinitesimal box. The volume element $\mathrm{d}V$ is used to account for neutrons that are lost or gained.

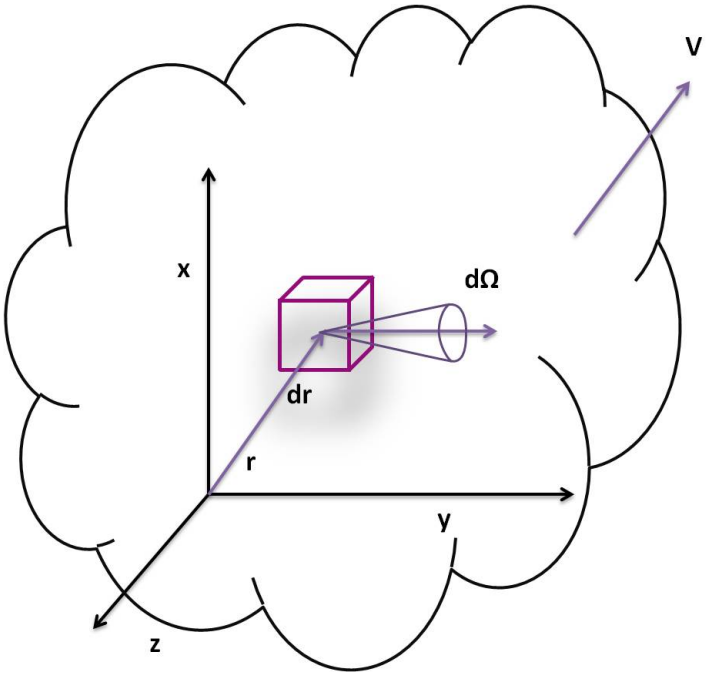


FIGURE 2.8: Particles in the infinitesimal box moving in a cone $\mathrm{d}\Omega$ in direction $\hat{\Omega}$.

Let the neutron density distribution of the infinitesimal volume at time t be defined as

$$N(r, \Omega, E, t) d\mathbf{r} d\mathbf{\Omega} dE dt,$$

then the neutron angular flux at time t is defined as

$$vN(r,\Omega,E,t)d\mathbf{r}d\mathbf{\Omega}dEdt \equiv \Phi(r,\Omega,E,t),$$

where v is the neutron velocity. The angular neutron current is the vector quantity made from the neutron angular flux by the unit vector in the direction of motion

$$J(r,\Omega,E,t) = \Phi(r,\Omega,E,t)\hat{\Omega}.$$

The time-dependent neutron balance in the infinitesimal volume dV is then:

$$N(r,\Omega,E,t+\Delta t) = N(r,\Omega,E,t) + P(r,\Omega,E,\Delta t) - L(r,\Omega,E,\Delta t), \tag{2.2}$$

where P is the neutron production and L is the neutron loss. The total number of neutrons in the volume V during time t to $t+\Delta t$ is $\Delta\Omega\Delta E\int_V drN(r,\Omega,E,t+\Delta t)$. The neutron gains in the system come from scattering and fission productions, and the losses are from the capture, inelastic scattering, and the leakage out of the volume V. Using equation 2.2, the neutron Boltzmann equation is as follows:

$$\begin{aligned}&\frac{1}{v}\frac{\partial\mathbf{\Psi}}{\partial t} + \hat{\mathbf{\Omega}}\cdot\bigtriangledown\mathbf{\Psi}(\mathbf{r},E,\hat{\mathbf{\Omega}},t) + \Sigma_t\mathbf{\Psi}(\mathbf{r},E,\hat{\mathbf{\Omega}},t)\\ &= \int_{4\pi}d\hat{\mathbf{\Omega}}'\int_0^\infty dE'\Sigma_s(E'\to E,\hat{\mathbf{\Omega}}'\to\hat{\mathbf{\Omega}})\mathbf{\Psi}(\mathbf{r},E',\hat{\mathbf{\Omega}}',t) + s(\mathbf{r},E',\hat{\mathbf{\Omega}}',t).\end{aligned} \tag{2.3}$$

The variables in Equation 2.3 are as defined in table 2.1:

The first term on the left-hand side is the time-rate of change in the angular neutron density. The second term describes neutron streaming (i.e. the rate at which neutrons travelling in the direction of motion $\hat{\Omega}$ leave and enter the differential volume element d^3r.) into and out of d^3r. The third term consists of all interactions removing neutrons from the flux either by absorption or scattering away from the differential angle, $\hat{\Omega}$ and energy, E.

Ψ	neutron flux
v	neutron speed
E	neutron kinetic energy
Ω	neutron solid angle (momentum direction)
Σ_t	total interaction cross section
Σ_s	the collision scattering cross section
s	the neutron sources (external and fission sources $$s_{external}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t)+$$ $$s_{prompt}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t) = \chi_p(E)\Sigma_i \int dE'\nu\Sigma_f \mathbf{\Psi}(\mathbf{r}, \mathbf{E}'\hat{\mathbf{\Omega}}', \mathbf{t})$$)

TABLE 2.1: Variable definitions used in the neutron Boltzmann equation 2.3

Equation 2.3 assumes that all the neutrons are emitted instantaneously at the time of fission. However, a small fraction of the neutrons are emitted later from fission precursors. These small fractions of neutrons become important in determining the time-dependent behavior of a nuclear reactor. So the source term in equation 2.3 can be defined as:

$$s_{tot}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t) = s_{external}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t) + s_{delayed}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t) + s_{prompt}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t). \tag{2.4}$$

Where $s_{delayed}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t)$ and $s_{prompt}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t)$ are

$$s_{prompt}(\mathbf{r}, E, \hat{\mathbf{\Omega}}', t) = \chi_p(E) \sum_i (1 - \beta_i) \int dE'\nu\Sigma_f^i(\mathbf{r}, E')\mathbf{\Psi}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t) \tag{2.5}$$

The delayed neutrons are emitted from many different fission products; however, for simplicity they are divided into six precursor groups with delayed yields (β_i) and decay

constants (λ_i). Therefore, the source of the delayed neutrons is defined as:

$$s_{delayed}(\mathbf{r}, E, \hat{\mathbf{\Omega}}', t) = \sum_l \chi_l(E)\lambda_l C_l(\mathbf{r}, t), \tag{2.6}$$

where $C_i(\mathbf{r}, \mathbf{t})$ is the concentration of the fission precursor of type i. Thus,

$$\begin{aligned}
&\frac{1}{v}\frac{\partial \mathbf{\Psi}}{\partial t} + \hat{\mathbf{\Omega}} \cdot \bigtriangledown \mathbf{\Psi}(\mathbf{r}, E, \hat{\mathbf{\Omega}}, t) + \Sigma_t \mathbf{\Psi}(\mathbf{r}, E, \hat{\mathbf{\Omega}}, t) \\
&= \int dE' \int d\hat{\mathbf{\Omega}}' \Sigma_s(\mathbf{r}, E' \rightarrow E, \hat{\mathbf{\Omega}}' \cdot \hat{\mathbf{\Omega}}) \mathbf{\Psi}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t) + s_{external}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t) \\
&\quad + \sum_l \chi_d(E)\lambda_l C_l(\mathbf{r}, t) + \chi_p(E) \sum_i (1 - \beta_i) \int dE' \nu \Sigma_f^i(\mathbf{r}, \mathbf{E}') \mathbf{\Psi}(\mathbf{r}, \mathbf{E}', \hat{\mathbf{\Omega}}', \mathbf{t})
\end{aligned} \tag{2.7}$$

Also the fission precursors are changing in time as they are decaying, and more fission products are added. This equation is as follows

$$\frac{\partial}{\partial t} C_l(\mathbf{r}, t) = \sum_i \beta_l^i \int dE' \nu \Sigma_f^i(\mathbf{r}, E') \mathbf{\Psi}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t) - \lambda_l C_l(\mathbf{r}, t). \tag{2.8}$$

The equations 2.7 and 2.8 are known as the kinetic transport equations.[5]

2.1.2 Criticality

To maintain a stable chain reaction in a nuclear reactor in time, the rate of neutron production needs to be balanced with the rate of loss. For this reason an important factor is defined for the reactors: the **"multiplication factor"**. This factor is defined as:

$$k_{\text{eff}} = \frac{P(t)}{L(t)} \tag{2.9}$$

[5] There are some assumptions made for the kinetic equations such as considering the prompt neutron spectrum (χ_p) and the delayed neutron spectrum (χ_d) are independent of each other [3]. This assumption may result in significant errors in the reactor kinetics. Another assumption is that the materials are assumed to be time-independent. Fortunately, both of these assumptions are generally true for reactor simulations or computational methods. [3].

To normalize the multiplication factor to zero, another quantity, the so called reactivity, is defined as

$$\rho = 1 - \frac{1}{k_{\text{eff}}} \tag{2.10}$$

The reactor is said to be in a critical state when $k_{\text{eff}} = 1$. In other words, when the number of neutrons produced is equal to the number of neutrons lost, the reactor is critical. When the rate of neutron losses and productions are not changing at the same rate in time, the neutron population may grow (supercritical) or decay (subcritical) exponentially according to the equation below 2.11[1]:

$$N(t) = N(0)e^{\frac{(k_{\text{eff}}-1)t}{l}}, \tag{2.11}$$

where k_{eff} is the multiplication factor and l is the neutron lifetime, which is defined as $l = N(t)/L(t)$ [1].

Considering the definition above, the three different states of criticality can be defined as below:

$$k = \begin{cases} < 1, & \text{Subcritical} \quad (2.12) \\ = 1, & \text{Critical} \quad (2.13) \\ > 1, & \text{Supercritical} \quad (2.14) \end{cases}$$

Figure 2.9 illustrates these states of the neutron population in time without any external neutron source. The solutions to equation 2.7 give no useful information on the criticality of the system as to whether it is subcritical or supercritical [3].

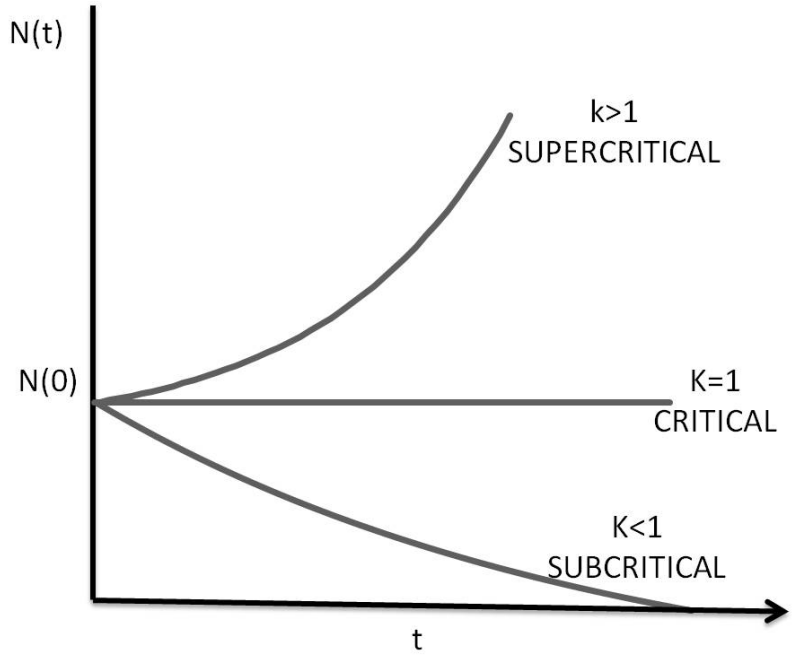


FIGURE 2.9: Time dependence of the flux for a source free multiplying medium [3].

2.1.2.1 Methods for Criticality Calculations

There are some methods to calculate the multiplication factor. The differences between the methods come from whether the time-dependent (static) or time-independent (dynamic) neutron transport equation is chosen. Following discussion describes and compares these methods very briefly.

Static criticality calculations

There are two methods to calculate criticality problems for the static case, and are known as k static and alpha static calculations. The common ground between these two static methods is that in both calculations, a parameter is introduced to adjust the system back to be critical [11]. The static solutions for criticality calculations solve the systems which are close to critical. While solving the static solutions to find the criticality of the system, the following needs to be considered:

- The solutions of the criticality calculations signify the asymptotic distribution of neutrons or the fundamental mode. (To achieve the neutron balance for steady

state of the system)

- External sources are neglected for simplicity;
- All neutrons are assumed to be promptly born;
- Media are considered to be time-independent (the material is not changing)

The k static criticality calculation

The k static calculation adjusts the neutron production to obtain a critical system. The time-independent neutron transport equation is achieved when $\frac{1}{v}\frac{\partial \mathbf{\Psi}}{\partial t} = 0$ in equation 2.3 and when introducing a factor k to account for balancing the neutron losses and productions in the system. Equation 2.15 is:

$$\begin{aligned} &\hat{\mathbf{\Omega}} \cdot \bigtriangledown \mathbf{\Psi}(\mathbf{r}, E, \hat{\mathbf{\Omega}}, t) + \Sigma_t \mathbf{\Psi}(\mathbf{r}, E, \hat{\mathbf{\Omega}}, t) \\ &= \int_{4\pi} d\hat{\mathbf{\Omega}}' \int_0^{\infty} dE' \Sigma_s(E' \rightarrow E, \hat{\mathbf{\Omega}}' \rightarrow \hat{\mathbf{\Omega}}) \mathbf{\Psi}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t) \\ &\quad + s_{external}(\mathbf{r}, E', \hat{\mathbf{\Omega}}', t) + \frac{\nu}{k}\chi(E) \int dE' \Sigma_f \mathbf{\Psi}(\mathbf{r}, \mathbf{E}', \hat{\mathbf{\Omega}}', \mathbf{t}) \end{aligned} \tag{2.15}$$

The k static method solves equation 2.15 to find the k factor in order to make the system critical. In other words, k is a factor that adjusts the number of neutrons per fission ν to preserve the balance between the losses and productions in the system, and eventually to achieve the criticality. Therefore, there should exist the largest value of k, which corresponds to the fundamental mode, to solve the equation 2.15 [3]. The k value then implies how much ν needs to be changed to keep the equation 2.15 in a balance. Thus $k < 1$ describes a subcritical system, and it requires an increase in ν to balance the equation 2.15. Similarly $k > 1$ represents a supercritical system, and to balance the equation 2.15, the ν has to be decreased.[3].

The k static method is applicable only for near critical systems, and this factor is not always the same quantity as k_{eff} (only at near critical systems).

The α static criticality calculation

The α static method solves equation 2.7 by assuming that the time and space components can be separated, as in equation 2.16 [3]

$$\mathbf{\Psi}(\mathbf{r}, E, \hat{\mathbf{\Omega}}, t) = \mathbf{\Psi}_a(\mathbf{r}, E, \hat{\mathbf{\Omega}})e^{\alpha t} \tag{2.16}$$

Inserting the equation 2.16 in equation 2.3, the following equation 2.17 is obtained

$$\begin{aligned}[\hat{\mathbf{\Omega}} \cdot \bigtriangledown + \Sigma_t + \frac{\alpha}{v}]\mathbf{\Psi}(\mathbf{r}, E, \hat{\mathbf{\Omega}}) = &\int dE' \int d\hat{\mathbf{\Omega}}' \Sigma_s(\mathbf{r}, \mathbf{E}' \rightarrow \mathbf{E}, \hat{\mathbf{\Omega}}' \cdot \hat{\mathbf{\Omega}}) \mathbf{\Psi}(\mathbf{r}, \mathbf{E}', \hat{\mathbf{\Omega}}') \\ &+ \chi(E) \int dE' \nu\Sigma_f(\mathbf{r}, \mathbf{E}') \mathbf{\Psi}(\mathbf{r}, \mathbf{E}', \hat{\mathbf{\Omega}}')\end{aligned} \tag{2.17}$$

The largest real component of α is a solution to 2.17 that corresponds to the fundamental mode solution [3]. The α values can be summarized to describe the different states criticality as

$$\alpha = \begin{cases} < 0, & \text{Subcritical} \\ = 0, & \text{Critical} \\ > 0, & \text{Supercritical} \end{cases}$$

In the subcritical systems, the α factor adds more neutrons with lower speed to the system in order to achieve the critical state. For the supercritical systems this factor removes the lower speed neutrons of the system for obtaining the criticality [3]. Since the α static method only considers the fundamental mode solution[6], this method is not useful for the transient behavior or higher order modes.

Dynamic criticality calculations

In dynamic criticality calculation method, the neutron losses and productions over a period of time(T) calculates the multiplication factor of the system

$$k_{\text{eff}} = \frac{N_P(T)}{N_L(T)}. \tag{2.18}$$

[6] Assuming that the system eventually relaxes to its fundamental mode which corresponds to a single exponential variation of the neutron distribution $N(t) = N(0)e^{\alpha t}$ [11]

This method makes no assumptions and finds the multiplication factor from its definition in equation 2.3. The neutrons are tracked over a period of time interval (T), and the material and simulation world geometry over this time interval are unchanged. [6]. Therefore, the rate of neutron productions (N_P) to neutron losses (N_L) is calculated at the end of the time interval. Table 2.2 illustrates briefly a comparison between the static and dynamic methods.

Methods	Accuracy	Time-dependency	Implementation
k-static	Very accurate only near critical	No	Easy
α-static	Very accurate near critical	Yes	Easy
Dynamic	Accurate at any state	Yes	Hard

TABLE 2.2: Comparison between the criticality calculation methods

Since the dynamic method has no assumptions, it is always applicable to calculate the actual k_{eff}. Where the static methods can only estimate whether a system is sub or super critical and give a rough estimate of how far from criticality the system is. This method is accurate only for near-critical cases.

Chapter 3

Computational Methods

In this chapter, two types of reactor physics computation methods –deterministic and Mont Carlo– are discussed. This discussion aims to fulfill the knowledge required for this project. Therefore, the deterministic method is briefly introduced, and only those Monte Carlo computer code used in this project are described more in detail.

Before proceeding to discussion of this chapter, an explanatory example is presented to demonstrate how the Monte Carlo and the deterministic methods actually work. The problem in this example is to calculate the area of a circle that is confined in a unit square as shown in Figure 3.1. It is known that the area of the circle is given by equation 3.1

$$A_{circle} = \pi r^2. \tag{3.1}$$

where r is the radius of the circle.

The deterministic method solves the area formula of the circle (3.1) by the given initial parameters, such as the radius of the circle and number of π. Then the exact value for area of the circle is calculated using this method and the outcome is the same for the same set of input parameters. The value for the area using deterministic method is to

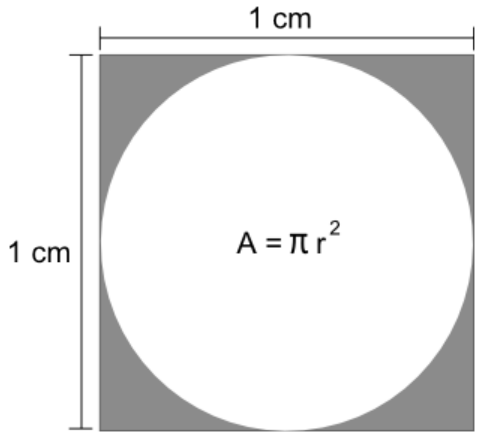


FIGURE 3.1: The circle confined in a unit square.

use equation 3.2

$$A_{circle} = \pi r^2 = \pi(0.5)^2 = 0.785. \tag{3.2}$$

The way that this problem is treated by the Monte Carlo method is by

1. Selecting a large number of points inside the square (Fig. 3.1):
   $(-0.5 < x < 0.5 \ \& \ -0.5 < y < 0.5)$.
2. Calculating the number of points that are falling inside the circle ($x^2 + y^2 < 1$);
3. Knowing the total number of selected points inside and the number of points falling inside of the circle, as shown in Figure 3.2. Their ratio is found as equation

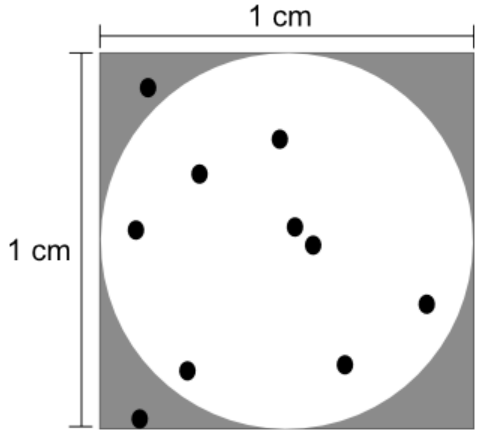


FIGURE 3.2: Monte Carlo random point selections

3.3

$$A = \frac{\bullet \text{ inside the circle}}{\bullet \text{ total}} = \frac{8}{10} = 0.8. \tag{3.3}$$

The procedures above should give the area of the circle. This procedure is repeated until the difference in results from consecutive runs are less than an uncertainty ($A_{i+1} - A_i < 0.0025$). Since the outcome of the simulation is always slightly different, this method of calculation is also called stochastic. As is seen in equation 3.3, the result is very close to the correct value. However, as the number of simulated points increases, the estimated result becomes much closer to the exact value found by the deterministic method.

This example is very simple, so that the use of the Monte Carlo method is unnecessary. However, many mathematical and physical problems have much convoluted equations that make the Monte Carlo method one of the most straightforward solution to them (e.g. the neutron transport). As mentioned earlier, the discussion of this chapter is focused mainly on the stochastical method (Monte Carlo method); since this project is built on using a Monte Carlo code tool set called Geant4. This tool kit will be discussed in section 3.4.

3.1 Deterministic Method

Deterministic neutronics methods solve the discretized neutron transport equation to obtain a system of equations for scalar flux [1]. To find solutions to the linearized Boltzmann transport equation, the seven-dimensional equation 2.3 (three spatial, two directional, energy, and time) needs to be solved analytically. All deterministic transport methods intend to find a flux solution for the neutron transport equation 2.3 and 2.7. The challenge of finding a solution is not in the complex mathematical formulation of the equation, but rather in the convoluted angular and energy dependency of the terms in this equation. Thus, some suitable approximations are introduced to simplify this equation and yet to preserve obtaining useful information of the average behavior of neutrons.

These simplifications can be obtained by replacing each continuous dimension: spatial, energy, and angular directions in equation 2.3 by a discretized set of function values at a discrete set of points [12]. This discrete set of functions converts the 2.3 equation to a system of algebraic equations which are solvable by the computers [12]. The following paragraph is a brief and short discussion on these discretizations.

The energy discretization is done by dividing the continuous energy into energy groups, which is also known as group structure. The next simplification is spatial meshing, where the simulation geometry is subdivided into smaller volumes V_n with homogeneous material properties. Another approximations to equation 2.3 is angular discretization. The outgoing angular distributions are generally chosen to be normal to the surface created by spatial meshing [6]. Discretizing the angular dependence can be done via different methods, as listed below [10]:

- The method of characteristics;
- The collision probability method;
- The discrete ordinate method;
- The method of spherical harmonics.

The discussion of the methods is out of the scope of this text. However, the diffusion method is discussed in the following section both for time-dependent and time-independent treatments of the neutron transport equation.

3.1.1 Diffusion Theory

Neutron diffusion theory is derived from the general transport theory. The diffusion equation is accepted as the simplest way to solve neutron transport problems. Before proceeding into deriving the diffusion equation, it is more convenient to work with a

balance equation for the scalar neutron density. This equation is as shown below:

$$\frac{1}{v}\frac{\partial}{\partial t}\Phi(r,E,t) + \nabla J(r,E,t) + \Sigma_t(r,E)\Phi(r,E,t) = \int_0^\infty [\Sigma_s(r,E' \to E)\Phi(r,E,t) + \frac{1}{4\pi}\chi(E)\nu\Sigma_f(r,E')\Phi(r,E',t)]dE', \quad (3.4)$$

Where $J(r,E)$ is the neutron current density.

The equation above was derived by integrating each term in equation 2.7 over the angular variable[1].

The next approximation to get to the diffusion equation is to integrate the equation 3.4 over energy variable to remove the continuous energy-dependence. Each term in equation 3.4 is integrated over some energy interval corresponding to energy group g (energy discretization). The integration of the terms is rather straightforward, and it is sufficient to replace the scalar flux $\Phi(r,E,t)$ by the scalar group flux $\Phi_g(r,t)$, the total cross section $\Sigma_t(r,E)$ by $\Sigma_{t,g}(r)$, and the neutron current density term $J(r,E,t)$ by $J_g(r,t)$. Therefore, the equation 3.4 with energy discretization becomes

$$\frac{1}{v_g}\frac{\partial}{\partial t}\Phi_g(r,t) + \nabla \cdot J_g(r,t) + \Sigma_{t,g}(r)\Phi_g(r,t) = \sum_{g'=1}^{G}[\Sigma_{s,g'}(r)\Phi_{g'}(r,t)] + \chi_g \sum_{g'=1}^{G}\frac{1}{4\pi}\nu\Sigma_{f,g'}(r)\Phi_{g'}(r,t)], \quad (3.5)$$

where

$$\frac{1}{v_g} = \frac{\int_{E_g}^{E_{g-1}} \frac{1}{v}\Phi(r,E)dE}{\int_{E_g}^{E_{g-1}} \Phi(r,E)dE},$$

$$\chi_g = \int_{E_g}^{E_{g-1}} \chi(E)dE,$$

and

[1] one can find a full derivation of the equation 3.4 in many text books such as [3] [1].

$$\Sigma_{s,g'\to g} = \frac{\int_{E_g}^{E_{g-1}} \int_{E_{g'}}^{E_{g'-1}} \Sigma_s(r, E' \to E)\Phi(r, E')dEdE'}{\int_{E_g}^{E_{g'-1}} \Phi(r, E')dE'}.$$

After removing the energy dependence on the neutron equation, it is time to show the actual diffusion theory. The diffusion theory suggests that the neutron current density $J(r,t)$ is proportional to the flux gradient, this approximation is known as Fick's law and is shown in equation 3.6 [1]

$$J_g(r,t) = -D_g(r)\nabla\Phi_g(r,t), \tag{3.6}$$

where $D_g(r)$ is known as the diffusion coefficient. For the diffusion equation to be accurate, it relies on the following approximations [10]

1. The angular flux is only weakly dependent on the angular variables;
2. The fission source is isotropic;
3. The time derivative of neutron current density is small compared to the flux gradient;
4. The anisotropic energy-transfer contribution can be ignored in group-to-group scattering.

Using the above approximations, the diffusion equation can be written as

$$\frac{1}{v_g}\frac{\partial}{\partial t}\Phi_g(r,t) - D_g\nabla^2\Phi(r,t) + \Sigma_{t,g}(r)\Phi_g(r,t) = \sum_{g'=1}^{G}[\Sigma_{s,g'}(r)\Phi_{g'}(r,t)] + \chi_g\sum_{g'=1}^{G}\frac{1}{4\pi}\nu\Sigma_{f,g'}(r)\Phi_{g'}(r,t)]. \tag{3.7}$$

Considering the discrete homogeneous material region, the spatial dependence of cross section and consequently the dependence of the diffusion coefficient is removed, and therefore

$$-\nabla D_g(r)\nabla\Phi_g(r,t) = -D_g\nabla^2\Phi(r,t) \tag{3.8}$$

As can be seen in equation 3.7, the diffusion equation depends on three spatial variables and time. Consequently, the solution methods largely depend on geometry. With discretization of space and energy, the set of diffusion equations must be individually solved, and the results of each equation are coupled together using suitable boundary conditions. The following two subsections demonstrate the algorithms that some computer codes follow to solve the static and kinetic diffusion equation respectively.

3.1.2 Static Deterministic Solution

A time-independent deterministic solution for the neutron transport equation is solved by using spatial mesh approximation, as was discussed above. Thus, the world geometry is divided into smaller volumes V_N to compute homogenized cross sections. Then the simulation world is a series of these spatial meshes, and the neutron flux is solved in each of these individual cells, and these solutions are coupled to one another with appropriate boundary conditions. The next step is to repeat these flux calculations iteratively until the results are within a reasonable range.

The neutron flux is determined at the center of each individual cell using so called a finite difference method. The name comes from the fact that the derivatives by finite-difference ratios are used. In this method, it is assumed that every cell has homogeneous properties. Figure 3.3 illustrates a full core in deterministic calculations. Every cell with the same color has the same homogeneous property (e.g. reflector (brown), outer-core (blue), inner-core (red)) [13].

The flux calculation starts with an initial guess of the flux values; then the diffusion equation is solved to find the value of the flux at the center of every cell. The divergence (leakage) term is solved by using the flux at midpoint of cell C (as shown in Figure 3.3) and the flux at the interface (the boundary between cell C and L or T, etc.). This calculation represents one iteration. This procedure is repeated for another iteration with using the previous calculated flux values. The next step is to repeat these flux

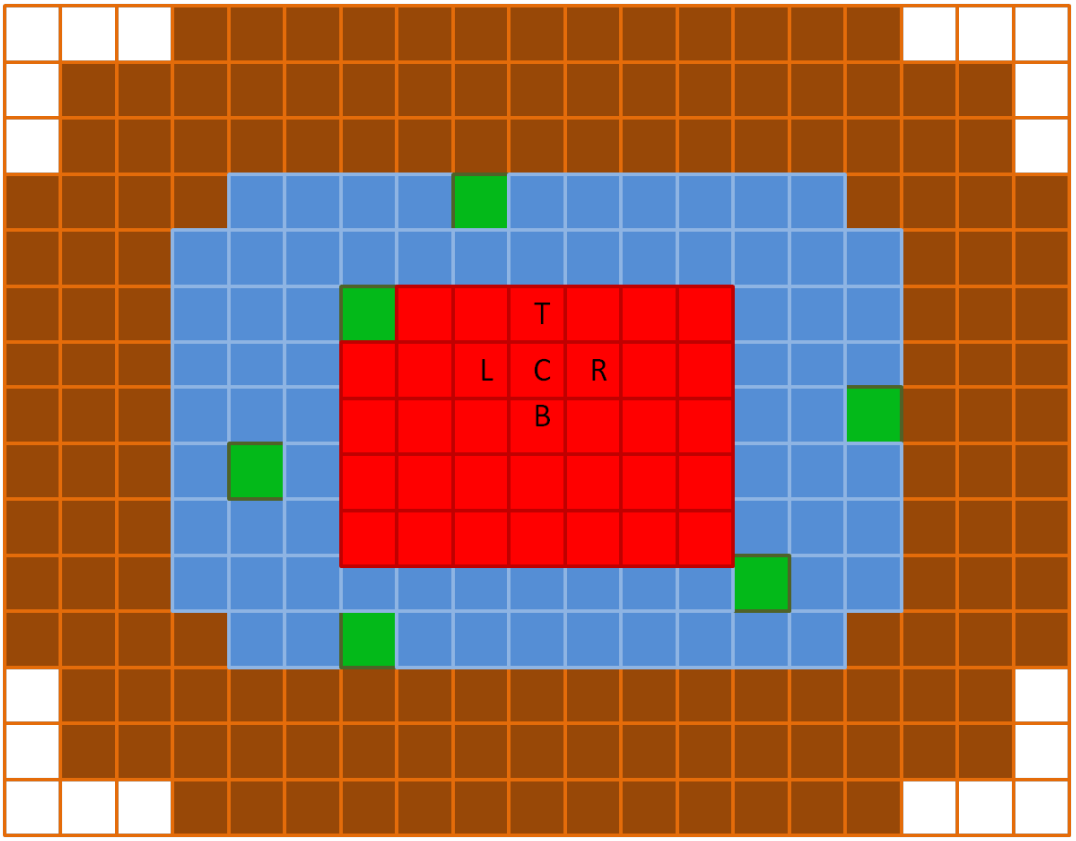


FIGURE 3.3: Geometry representation of full core defined in a deterministic code. Adapted from Dr. Rouben's lecture note

calculations iteratively until the results are within a reasonable range. The value of the flux is said to be converged when the ratio of the flux in the current iteration to the that in the previous iteration at every cell in the geometry is close to 1 [13]. The following chart 3.4 exemplifies the algorithm of this method .

3.1.3 Dynamic Deterministic Simulation

In the time-dependent case, the neutron equation has a partial differential term in time compared to the steady-state case that is only the ordinary differential equation. Finding the solution to the time-dependent analogue requires additional approximations to simplify the time dependence term [14].

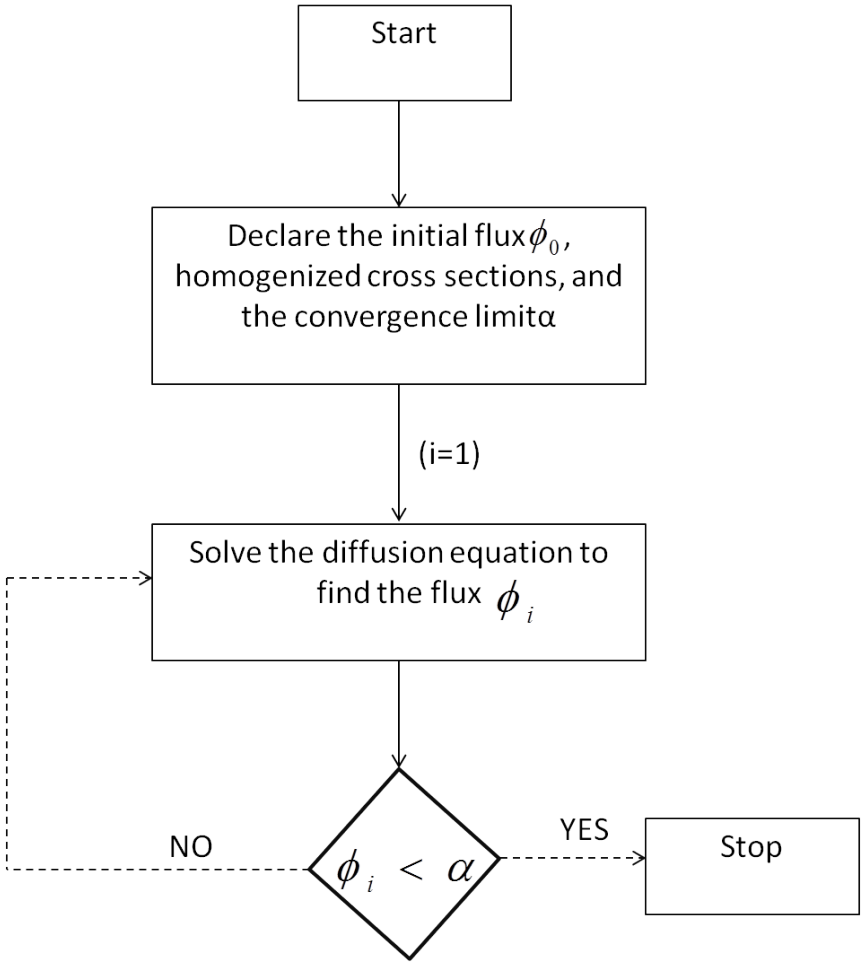


FIGURE 3.4: Flow chart of iterative flux solution.

There are different approaches to solve for the time-dependent transport equation. The methods to solve the dynamic transport equation can either use point-kinetics approximations or some space-energy dynamics methods, such as the finite difference method, the modal and nodal approaches, and quasi-static method. Some of these methods are described briefly below.

The point kinetics approximations assume that the solution to this equation is a separable solution into a space-energy function (Φ time-independent neutron flux) and time function ($T(t)$) as in equation 3.9.

$$\Psi(r, E, t) = \Phi(r, E)n(t). \tag{3.9}$$

Thus, using this assumption, the point kinetic equations 2.7 and 2.8 become [15]

$$\begin{cases} \dfrac{dn(t)}{dt} = \dfrac{\rho(t)-\beta}{\Lambda(t)} n(t) + \displaystyle\sum_{i=1}^{n} \lambda_i C_i(t) & (3.10) \\ \dfrac{dC_i(t)}{dt} = \beta_i \dfrac{\rho(t)}{\Lambda(t)} n(t) - \lambda_i C_i(t) & (3.11) \end{cases}$$

where the variables used in equation 3.10 and 3.11 are as in table 3.1

ρ	the reactivity
Λ	the average neutron generation
β_i	the delayed neutron fraction of group i (n is the number of these percurosrs)
λ_i	the delayed neutron decay constant
C_i	the delayed neutron precursors concentration

TABLE 3.1: Variable definitions used in the point kinetics approximations

The point kinetics approximation approach faces many restrictions as in the following

- Equations 3.10 and 3.11 do not consider the energy effects.
- This method uses an average over the delayed neutrons characteristics (i.e., β_i and λ_i) for each isotope.
- This method assumes a static flux shape which is only true for the system with static material compostions.

To eliminate some of these restrictions, the space-energy dynamics methods were introduced. These methods are defined in the table 3.2 [15]. However, these approaches have some limitations because of the discretizations of space, energy, and time that affect the accuracy and speed of the calculations.

3.2 Monte Carlo Method

Monte Carlo computer codes use the stochastic method to estimate the solutions of the neutron transport equation. The advantage of this method is that the necessary

Methods	Description
The quasistatic method	The quasistatic method solves the flux shape in regular time steps using the equation 2.7.
The finite difference method	The finite difference method solves the equation 2.7 with finite difference quotients for the neutron flux across space-energy meshes at each time step.
The modal method	The flux is represented as a superposition of fundamental space-energy modes, where the time-dependent superposition coefficients are updated at each time step using this method.
The nodal method	The neutron flux is represented as individual coupled fluxes at spatial nodes, where the time-dependent coefficients are updated at each time.

TABLE 3.2: The space-energy dynamic approaches

approximations in the deterministic method are superfluous for finding the solutions for a complex system such as nuclear reactors. Thus, the Monte Carlo method is an ideal candidate for solving three-dimensional, time-dependent problems. Consequently, the continuous treatments of all degrees of freedom in neutron transport equation preclude the discretization errors in the calculation, and the error in Monte Carlo calculations turn up as the stochastic uncertainty [3].

The Monte Carlo method is based on a random sampling scheme to solve a physical problem [16]. Unlike the deterministic methods that solve an exact equation, rather Monte Carlo estimates the solution[2] by tracking each individual particle and recording some features of their average behavior [17].

Monte Carlo simulations start with generating a finite number of particles (N) in the medium of study, that is modeled in the computer, as if coming from a source. Each of these particles is tracked, and the history of the events[3] in which they participate is recorded. The behavior of this system is described by the probability density function (PDF) of particles in space[4]. Each particle's history is constituted from the events,

[2] Monte Carlo provides information about some specific physical quantities (tallies) which are requested by the user [17].

[3] Each event refers to a specific neutron interaction (collision, absorption, fission, escape, etc.)

[4] This equation turns out to be the same as the integral transport equation[17].

which are obtained by random sampling from the PDF's. Suppose that the purpose of these simulations is to obtain the desired result of finding quantity x. Then this purpose is fulfilled as below

$$x' = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{3.12}$$

The uncertainty in x' decreases as the number of histories (N) increases, which in most cases is proportion to $N^{\frac{1}{2}}$.

Thus, calculating a random variable using the Monte Carlo method requires one defining two functions: the cumulative probability distribution function and, as was mentioned above, probability distribution function. These two functions are essentially describing how probable it is that this random variable x' occurs in an interval between x and $x + \Delta x$ (PDF), and this random variable x' is less than or equal to x (CPD) [3]. The PDF is defined as $f(x)$

$$f(x)\Delta x = P\left\{x < x' < x + \Delta x\right\}. \tag{3.13}$$

The cumulative probability function is defined by

$$F(x) = P\left\{x' < x\right\}. \tag{3.14}$$

It is important to note that often the random variable x' (e.g. scalar flux) is not the random numbers that are sampled from the particle's history. Therefore, the properties (e.g. the mean number of collisions $\bar{c}$ in some volume V with a total cross section σ) must be sampled from the simulations that are necessary to calculate x' ($\Phi = \frac{\bar{c}}{V\sigma}$) [3].

3.2.1 Neutron Transport Simulation in Monte Carlo

In the Monte Carlo method, a number of initial neutrons are generated from a source[5] and are moving based on functions of probability of the neutrons' interactions through the medium of the study. These neutrons then are tracked from birth (e.g. fission or neutrons source) to death (e.g. absorption or capture). The neutrons move through a series of steps that end with an interaction. The type of the neutron interactions and the path length depend on the cross sections of the materials in which they are traveling.

Thus, according to the local material in the system of study, the neutrons undergo the possible interactions which may cause either the creation of the secondary neutrons or the disappearance of the initial neutrons. Therefore, each initial neutron has a history that consists of information about the number of the secondary neutrons, positions, and time of occurrence of interactions. An event is then the history of the initial neutrons and their secondary neutrons from creation to loss [6].

3.2.1.1 Monte Carlo Simulation World and Initial Source Distributions

The simulation world for a particular study requires all the geometries and their material compositions to be known and defined in the Monte Carlo simulation world. The simulation world consists of one mother volume that includes all the daughter geometries in the design. The mother volume is marked as the largest volume in the simulation world, and outside of this volume the probability of any interaction is zero. Thus, when a neutron has crossed the world's boundary, it is considered as a loss. The next largest volume in the mother world may contain many smaller geometries or daughter volume inside it. Also each volume can be divided into smaller volumes (also known as meshes) for finding more accurate results for scoring. To define the simulation world, some restrictions are required as follow:

[5]There are various kinds of neutron sources used in Monte Carlo simulations such as point source, uniform source, distributed source, etc [6]

- No overlap is allowed between the daughter volumes. The reasoning behind this restriction is that when the particle is in the overlapping region, the simulation cannot decide which geometry and material compositions to choose.

- The number of meshes in a volume is required to be as small as the minimum dimension that the simulation code can resolve. As the number of meshes increases, the number of boundaries that needs to be checked for determining which volume the neutrons are in would increases [6].

The primary neutrons can be generated in different forms at the beginning of the simulation, as described in table 3.3 [6]. The initial neutron distribution will converge in

Neutron initial source distribution	Description
Beamline	starting at the same positions and the same momentum direction.
Point source	starting at the same position but random momentum direction.
Uniform source	starting at positions uniformly distributed across the simulation world.
Distributed source	starting both position and momentum direction according to a given distribution.

TABLE 3.3: Common initial source distributions

time and space to the physical distribution. In fact, the neutron primary source can start at any distribution, but applying some of the initial neutron distribution 3.3 may converge more quickly than others. For instance, the distributed initial source is the fastest to converge toward the true physical spatial distribution. The source convergence is a significant factor for some calculations, such as criticality calculations. In criticality calculations, the rates of the neutrons losses and productions do depend on spatial distribution of the neutrons. Therefore, the neutron source has to be converged before any solution estimation can be made [6] [18].

3.2.1.2 Monte Carlo Calculations

After the primary neutrons are generated, each individual neutron is tracked in the simulation world. When tracking the neutron, the following information needs to be known:

1. The current position of the neutron;
2. The material M in the volume in which the neutron resides;
3. The material compositions (finding the elements and isotopes that are used in the material);
4. The total cross section in the material ($\Sigma_t = \sum_k N^k \sigma_t^k$, where σ is the microscopic cross section).

Once the above information is determined, the simulation will be done through the procedure that is described in Figure 3.5. As shown in Figure 3.5, the first step is to find

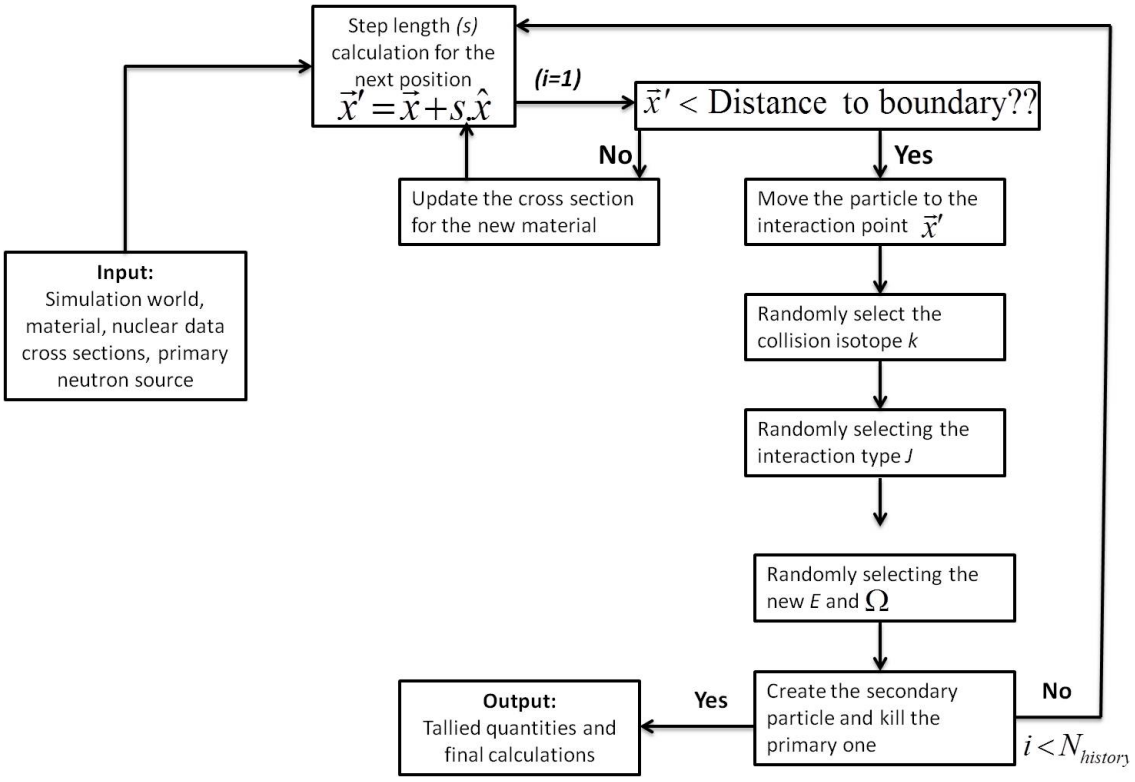


FIGURE 3.5: Data processes in a Monte Carlo simulation.

the new position of the neutron. The step length s can be randomly sampled by knowing

the total macroscopic cross section of material M (Σ_t). The PDF for distance s is

$$f(s) = \Sigma_t e^{-\Sigma_t s}. \tag{3.15}$$

Using the equation 3.15, the sampling procedure is

$$F(s) = 1 - e^{-\Sigma_t s} \rightarrow s = -\frac{ln(\xi)}{\Sigma_t}, \tag{3.16}$$

where $\xi \in [0, 1)$. If the distance s is less than the distance to the boundary, the particle is moved to the interaction point. The next step is to randomly select the collision interaction probability at the interaction point. The collision isotope k is sampled such that

$$P_{k-1} \leq \xi < P_k, \tag{3.17}$$

where P_k is the discrete cumulative distribution function that is defined as:

$$P_k = \sum_{i=1}^{k} \frac{N^k \sigma_t^k}{\Sigma_t} \tag{3.18}$$

After randomly selecting the collision isotope, it is time to select the reaction type. To sample the reaction type J for isotope k follows the same procedure as sampling the collision isotope k with a probability function defined as:

$$p_J = \frac{\sigma^J}{\sigma_t}, \tag{3.19}$$

where $\sigma_t = \sigma_{elastic} + \sigma_{inelastic} + \sigma_{capture} + \sigma_{fission}$. After the above sampling procedure is done, the exit energy and direction of the neutron are determined. If the secondary neutrons are created at this new location, then the primary neutron is removed from the simulation. This procedure is known as one **history** and at the end of each history some of the physical quantities, requested by the user, are tallied and written out. This procedure is repeated for more iterations until the tallied quantities are converged.

The outcomes in the Monte Carlo simulations come from tallying and scoring processes. The results from the tallies created by the scoring processes are raw data at the end of each simulation. To produce useful information from these raw data, these data are used to calculate some useful physical quantities such as k_{eff}. For instance, the multiplication factor of the system is calculated by tallying the number of the neutrons at the beginning and the end of every cycle throughout the simulation. Figure 3.5 illustrates the criticality calculation in MCNP.

There are several methods used to calculate the criticality in Monte Carlo simulations. The k-static method, the α-static method, and the dynamic method, which were discussed in details in section 2.1.2, are briefly described in Table 3.4.

criticality calculation methods	Descriptions
k-eigenvalue method	The ratio between the number of neutrons in successive generations [17] (time-independent process). $k_{generation} = \frac{\text{Number of neutrons i+1}}{\text{Number of generations i}}$ [6]
α-eigenvalue method	The multiplication factor is calculated by $k_\alpha = \alpha T_R + 1$ Where $\alpha = \frac{1}{t_2 - t_1}[\log(\frac{N(t_2)}{N(t_1)})]$ and T_R is neutron time removal [11]
Dynamic method	The criticality of the system at time t is calculated by $k_{\text{eff}} = \frac{N_P(T)}{N_L(T)}$ (3.20) Where T is the interval $[t_0, t]$, and N_P and N_L are the total neutrons produced and lost over period T.

TABLE 3.4: Criticality calculation methods in Monte Carlo simulation

3.2.2 GEANT4 Monte Carlo Toolkit

Geant4 is an open-source object-oriented simulation toolkit that has capabilities to accurately track the particles through matter. The Geant4 toolkit was developed by the collaboration of many scientists and engineers worldwide. Geant4 is implemented in the C++ programming language, which is an object oriented computer language. The object-oriented method made it easier to effectively manage complexity and limit dependencies by defining a uniform interface and common organizational principles for all physics models [19]. This toolkit consists of functions and classes that encompass every aspect for physics simulation process, including tracking, geometry, physics models and hits [19]. This software system has capabilities to handle everything from simple to complex detector geometries, and available physics models cover a wide range of energy for the interactions of particles with matter.

3.2.2.1 History

Geant4 development first appeared in 1993, when two independent studies at CERN and KEK were investigating how the modern computing techniques can improve the existing GEANT3[6] program [19]. The two studies then merged and it was decided to develop a simulation program based on object-oriented method using C++ computer language [21]. The GEANT4 project (RD44) became an international collaboration between Europe, Russia, Japan, Canada, and the United States.

The first release of Geant4 was completed in 1998 [21], and the Geant4 collaboration was established for continuing on the development, and granting maintenance and user support in 1999 [19]. This collaboration project is managed in a hierarchical organization divided into three groups: A Collaboration Board which is responsible for managing the resources and agreements for the Geant4 project, a Technical Steering Boarding

[6] The GEANT3 program was a software for particle physics interactions simulations thorough matter using Monte Carlo method implemented in FROTRAN [20].

which is responsible for the decision making about the manner in which physics phenomena are implemented, and the working groups which are working individually on the maintaining and developing of the physics phenomena code libraries [19].

Geant4 is freely available with comprehensive installation documentation, user and reference guide, and training kit. Also, several online code browsers exist for helping users navigate the source code. The collaboration also runs an online user forum with subforums according to different areas of interest [19].

3.2.2.2 Structure

Geant4 has been built around the concept of building a toolkit as the basis for the simulation components[7]. In other words, this toolkit provides a diverse range of flexible and modular components which enable the user to pick only those components needed for one's purposes [19]. The general hierarchical structure and data progression flow in Geant4 is shown in the flow chart 3.6. As one of the requirements for this toolkit, the category dependency has unidirectional flow, meaning that the upper hierarchical categories have no dependencies in any object in the lower hierarchical groups.

Figure 3.6 illustrates the categories of the classes for the major components of the Geant4 toolkit. The categories at the bottom of Figure 3.6 are the foundations of this toolkit and are used by the higher categories [22]. These categories are listed and briefly described in Figure 3.7 [22]. Before proceeding to the next section, note Table 3.5 which represents the definitions of those classes that are essential for describing the tracking procedure in Geant4 [6].

[7] The Geant4 toolkit is composed of three major parts: the source code, the nuclear data, and the utility files

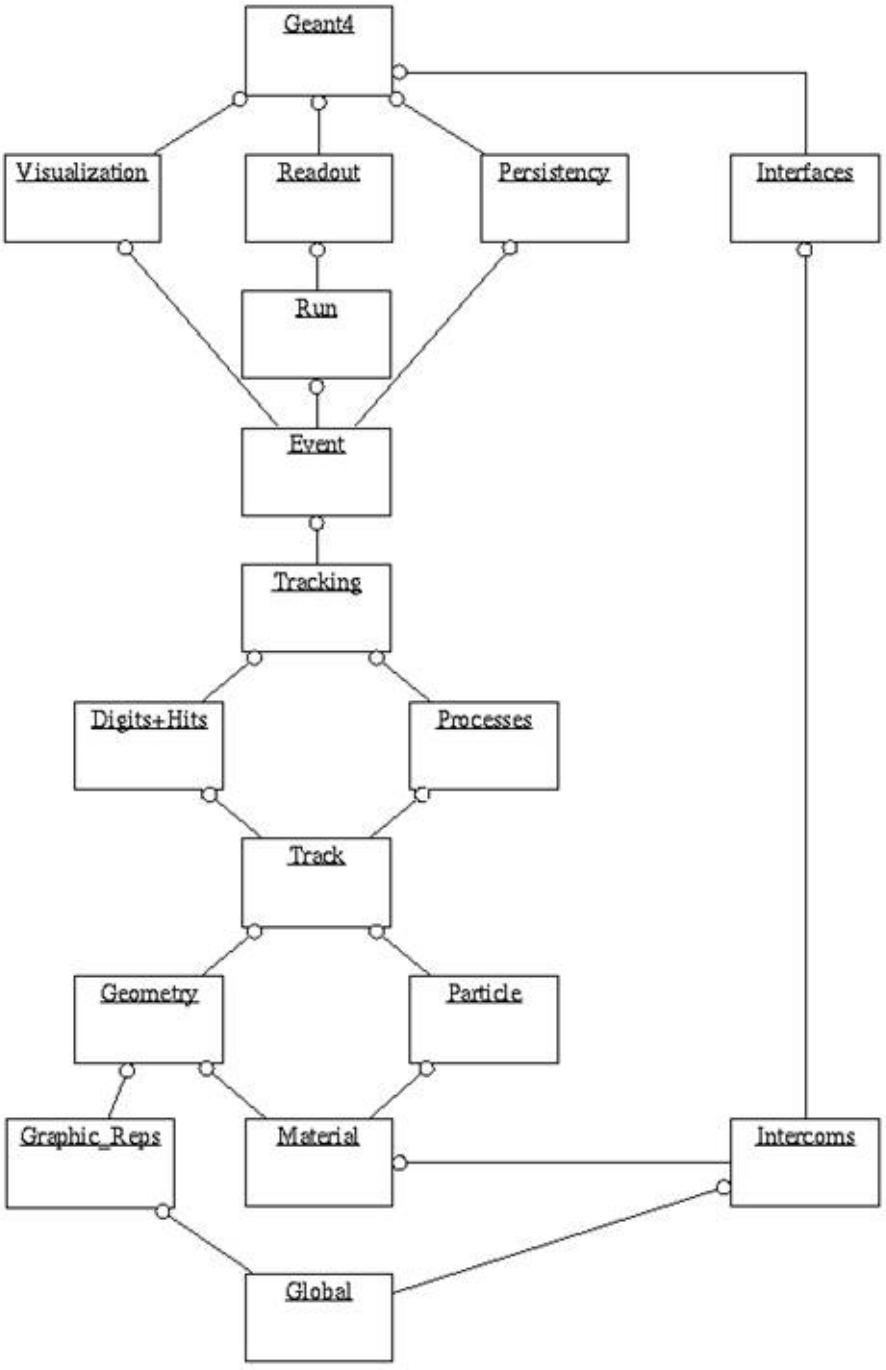


FIGURE 3.6: Geant4 simulation hierarchy.

The tracking manager is responsible for passing the necessary messages between the upper hierarchical object and lower hierarchical objects of the class categories that they manage [19]. Their task is to carry out the necessary actions in the tracking process by accessing the specific class that they manage. Some of these manager classes are:

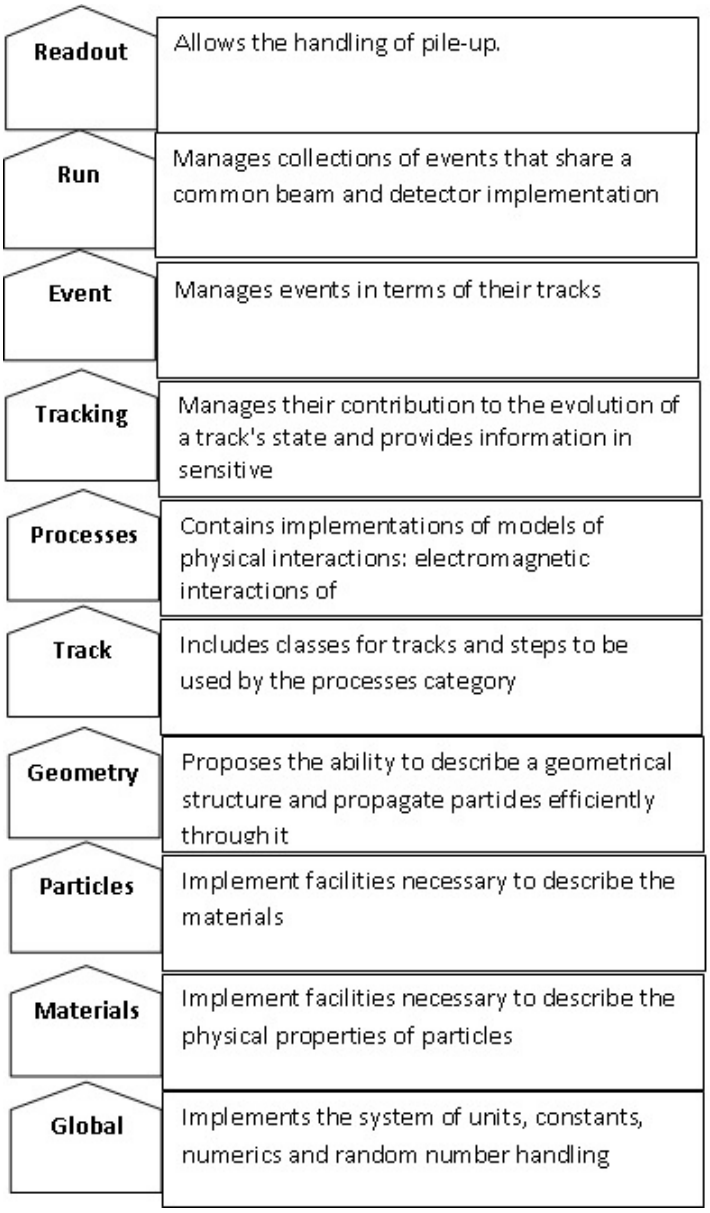


FIGURE 3.7: The definition to each class category in Geant4 simulation hierarchy.

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

Classes	Definitions
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.
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.
Run	A collection of independent events.
Simulation	The entire modeling process from start to finish, which may include multiple runs.

TABLE 3.5: Definitions of some of the Geant4 classes

To use the Geant4 toolkit for building an application, three mandatory classes and a main driver function must be implemented by the user:

1. G4VUserDetectorConstruction: Defining the material and geometry of the simulation world[8]

2. G4VUserPhysicsList: Defining the physics process to be used in the simulation[9]

3. G4VUserPrimaryGeneratorAction: Defining source distribution of primary particles

4. Main driver file: containing the main function to instantiate the kernel, the run manager, and the user defined classes.

There are some optional classes that the user may implement such as:

Run Action

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

[8] Physical environment that the particles move through during the simulations (i.e., a reactor geometry and its material compositions)

[9] The physics interaction between the particles through matter (i.e., neutrons interactions with matter)

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.

Although the above classes are implemented by the user, the functions are defined in the Geant4 toolkit, and the user can choose from the Geant4 built-in classes and functions to quickly create the user classes.

3.2.3 General Geant4 Simulation Scheme

Simulations in Geant4 consist of N_{Run}, and each run has N_{Event}. Every event is made up of several number of tracks (N_{Track}), and every track is composed of steps that propose all the physics processes associated with each individual particle [19].

At every stage from the interactions (hit) in the step level to the run level, the data is collected and analyzed. The data processing in Geant4 can be categorized into two group actions: initial actions and final actions.

3.2.3.1 Initial Actions

Geant4 tracks the particles as they go through the physics processes step by step. The tracking in Geant4 handles a physics process with the use of three actions:

- At rest, when interactions occur on a particle at rest.
- Along a step, when the interactions occur along the step.
- Post-step, when the interactions occur at the end of the step.

Each physics process possesses a step length for the particle using these actions. The step length calculations depend on the physics process. For the processes which depend on nuclear data such as hadronic processes, the step length is found as:

$$d_i = \frac{\eta_i}{\Sigma_i} \tag{3.21}$$

Where the variables of equation 3.21 are defined in Table 3.6.

d_i	step length associated with hadronic process i
Σ_i	the cross section associated with process i
η_i	the number of interaction lengths left $\eta_i = -\log(r)$ $r \in (0, 1]$

TABLE 3.6: Parameters definitions for equation 3.21

Therefore, each track starts by calculating the η_i using equation 3.21. This value is updated when any characteristics of the particle change [6]. The appropriate interactions take place at the pre-step, along-step, and post-step. Also, the process with smaller step length is chosen while every process proposes a step length [6]. Each interaction (hit) is scored and saved in a so-called object hit, which is a container class. Only those interactions are registered that occurred in the volumes which are flagged by a scoring class (i.e., sensitive detector class). The saved data from the hit collection will then be analyzed by the event action and the analyzed information will then be passed on to the run action. The event action also finishes all the tasks which need to be done at the beginning or end of each event.

3.2.3.2 Final Actions

As was mentioned earlier in this section, every run is composed of several events. Therefore, the run action will collect all the analyzed data from the event actions and will complete all actions at the beginning and end of each run. If the simulation consists of several runs, the data processing ends at the simulation level by analyzing data from every run.

3.3 Nuclear Data

The results of any reactor physics simulation codes are significantly dependent on the nuclear interaction data that is used in the simulations. The nuclear data used in the reactor physics simulation codes includes the necessary information to model the neutron interactions. These parameters are such as, the incoming energy of the an incident neutron, interaction cross sections, outgoing angular and energy distributions, and the secondary particle yield [6].

All the nuclear data is derived from certain evaluation organizers that are called *evaluated nuclear data libraries*. The information of the interactions between the incident neutrons and the target nuclei is based on experimental measurements and theoretical nuclear models [10]. The nuclear cross section data is evaluated at a certain temperature, often at zero Kelvin.

A change in temperature affects the velocity of the nucleus and consequently the nucleus energy. The nucleus velocity can be defined as a Maxwell-Boltzmann distribution. The velocity distribution of the nucleus, which is characterized as an ideal gas in thermal equilibrium, is defined as follows

$$F(v) = N(\frac{M}{2\pi kT})^{\frac{3}{2}} e^{-\frac{Mv^2}{2kT}} \tag{3.22}$$

where N is the number of particles, M is the mass of the nucleus, k is the Boltzmann constant, v is the velocity of the nucleus. Therefore, the average nuclei have a greater speed as temperature increases.

There are resonant peaks in the cross section data as discussed in section 2.1.1.1, and changes in the temperature causes changes in the target nucleus velocity. Thus, the neutrons that did not have the proper energy to form a compound nucleus, may now have enough energy to cause nucleus formation [6]. Therefore, the changes in the temperature broaden these resonant peaks. This effect of temperature on the resonant peaks is called *Doppler broadening*. Doppler broadening of the data library can be applied before the simulation using some codes such as NJOY. Also, they can be done on-the-fly as the temperature of the material is changing.

Depending on the simulation codes, the format of these data libraries may vary. These differences may come from the way of treating Doppler broadening of the cross sections and/or from the layout of the cross section tables. For example, the MCNP simulation code uses the data format which differs from the Geant4 data libraries. In Geant4 simulations, the MCNP data libraries cannot be used unless they are converted to the nuclear data library in Geant4, known as G4NDL[10].

[10]Geant4 Nuclear Data Library

Part II

Modelling the Experiment in the G4STORK code

Chapter 4

Previous Related Research

The emphasis in this chapter is on the literature of the most recent related research to this study. Described briefly are only references that are considered to be of general interest. This discussion required a judgment decision, and there may be more related research that are omitted or have not been described properly. Therefore, the more easily accessible literature is covered in this chapter.

The first section focuses mainly on similar subcritical measurements experiments and briefly discusses this research. The second section discusses the related time-dependent Monte Carlo simulation codes and points out the differences between those and G4STORK code.

4.1 Experiments Related to Subcritical Measurements

Naing et.al. [23] performed a number of experiments in a Westinghouse pressurized-water reactor type 2-loop plant.[1] The purpose of this study was to investigate the applicability of the modified Neutron Source Multiplication (NSM) method with extraction

[1] Cycles 7 and 8 of Tomari Unit No. 2

of the fundamental mode. The measurements were collected regarding a withdrawal of a sequence of control rod banks during the reactor startup.

The subcriticaclity of the reactor was achieved by insertion of control rods in various patterns during the start up [23]. The subcriticality was established using the inverse count rate (M) method. The modified NSM method, as defined in equation 4.1, was used to estimate the subcritical reactivity for these measurements.

$$\rho_l^s = C_l^{im} C_l^{sp} C_l^{ext} \rho_{ref}^s \frac{M_{ref}}{M_l} \tag{4.1}$$

Where all the variables above are defined in Table 4.1. The corrections ($C_l^{im}, C_l^{sp}, C_l^{ext}$)

criticality calculation methods	Descriptions
ρ_l^s	Estimated subcriticality of l-th subcritical state
C_l^{im}	Correction factor to the disturbance of neutron importance field
C_l^{sp}	Correction factor to the spatial effect caused by the disturbance of the fundamental mode
C_l^{ext}	Correction factor to the extraction of the fundamental mode
ρ_{ref}^s	The reference subcriticality
M_{ref}	The reference state
M_l	The count rate of neutron detector of l-th subcritical state

TABLE 4.1: Definitions of the variables in equation 4.1

were evaluated from numerical analyses of eigenvalue, and fixed source problems that were defined by three-dimensional diffusion equation 3.4 were computed using the finite difference method (as discussed in section 3.1.2).

Naing et. al.'s experimental research is similar to the subcritical measurements in ZED-2 in some ways; for example, both experiments were done for subcritical measurements. However, they differ in many ways:

1. The experiments were done in different kinds of reactors (Naing et. al. was done in light water (PWR) and Atfield et. al. was done in a heavy water reactor)

2. The subcriticalities were evaluated using different methods (Naing et. al. used a modified version of NSM method, and Atfield et. al. used Inverse Point Kinetics method)

4.2 Codes Related to G4-STORK

Before going into details and discussing G4-STORK in the next chapter, some computer codes related to G4-STORK and some previous studies on developing computer codes similar to G4-STORK are introduced. There are many computer codes that are specifically used in reactor physics analysis such as MCNP, WIMS, SCALE, PARK, DRAGON, DONJON, and many more. However, only those codes that are either used to validate G4-STORK results for this study (such as MCNP or codes that have more similarities to G4-STORK, such as time-dependent Monte Carlo codes) are described in this section.

4.2.1 MCNP5

MCNP is a three-dimensional, continuous-energy Monte Carlo simulation code that was developed by Los Alamos National Laboratory [17]. The MCNP code is written in the FORTRAN computer language, and includes both geometry and output tally plotters [17]. MCNP features the MCNPX versions that are the extended series of the main MCNP versions. Theses series consist of the features that were not included in the main series, such as exotic particle tracking (muons and neutrinos), interchangeable physics model, and some burnup calculations [6].

MCNP can be used for transport of neutrons and other particles (electron, photon, and coupled neutron-photon-electron) [17]. Also, MCNP has the capability to calculate eigenvalues for critical systems [17].

Thus, MCNP is mainly used for physics simulations and criticality calculations for near critical states. MCNP solves the criticality problems using the k-eigenvalue method that was described in Table 3.4. As mentioned earlier, the k-eigenvalue method is the ratio between the numbers of neutrons in successive generations and is a time-independent approach, which is only accurate for near critical regime [6].

MCNP has been widely used for validating the deterministic codes for the criticality calculations and other characteristic properties. Since MCNP finds the solutions without applying those necessary simplifications, as in deterministic codes such as energy discretizations, MCNP provides more accurate results than those from the deterministic codes.

While MCNP is the most widely used transport calculation code based on the Mont Carlo method, it fails to perform the criticality calculations for non-critical cases. Also, implementing new behavior is practically impossible, since MCNP source code is not available to be accessed by the user, unlike the G4-STORK code that is based on Geant4 toolkit that grants its users an easy access to flexible source codes.

4.2.2 TART 2012

TART is a three-dimensional, time-dependent, coupled neutron-photon Monte Carlo transport code [24]. TART was developed by Lawrence Livermore National Laboratory. The first version of this code was released in 1995 [24]. The subsequent versions of this code have been released at regular periods until the latest version, TART 2012. TART is written in standard FORTRAN computer language; however, the graphics portion of the system requires a C compiler [24].

TART versions have been improved in the physics, the nuclear and atomic data used by the code. The previous versions of this code used the multi-group, unshielded cross section until the TART 2005 version that added the continuous energy cross sections

feature to this code. While using the multi-group cross sections feature may save the computational simulation time, it only decreases the computational time by approximately a factor of two compared to using the continuous energy cross section. Thus, there is not much benefit to using multi-group cross section library [24].

TART is capable of performing the criticality calculations in three methods that were described in Table 3.4. TART uses a periodic renormalization of the neutron population to keep the population within a manageable range for calculating the criticality using the dynamic method. The dynamic method is used in G4-STORK for calculation of k_{eff} which is described later in chapter 5.

Although G4-STORK and TART share many fundamental concepts in dynamic reactor physics calculations, TART does not support dynamic material or geometric changes in time [6].

4.2.3 Serpent

Serpent is a three-dimensional, continuous energy Monte Carlo reactor physics burnup calculation code [25]. This code was developed by VTT Technical Research Center of Finland since 2004. This code is written using ANSI-C language, and uses GD open source graphics library to create the graphical output [25].

The first version of this code (Serpent 1) was developed in a short time, mainly to develop a Monte Carlo neutron transport code for reactor physics calculations at the fuel assembly level [10]. As the work proceeded, adding more calculation routines (such as burnup calculations to the source code) caused some problems, such as dealing with extensive memory usage and complicating the source code with adding more routine calculations. The Serpent2 has the capability to calculate burnup for fuel assembly in two dimensional geometry to the full-core problems in parallel mode simulations.

Serpent is a Monte Carlo code that has many capabilities such as [25]:

- Calculations of spatial homogenization and group constant generation for deterministic reactor codes
- Burnup calculations for the detailed assemblies for fuel depletion studies

Calculations of various reactor physics parameters (such as criticality calculations using k-eigenvlaue method) at pin, assembly and core levels to model any critical reactor type.

Considering all the applications of Serpent, this code can only calculate the near-critical systems. Unlike Geant4, Serpent is not an open source code [26]. Therefore, it lacks the flexibility and accessibility to modify the source code for implementing calculations as users desire.

Chapter 5

Methodology

As was mentioned in chapter 1, the G4-STORK code can be defined as a three-dimensional, continuous-energy Monte Carlo neutron transport code. Although this code is based on the GEANT4 tool kit, some extensions were added and modified to be usable in reactor physics calculations. The main work of this study was to model the ZED-2 subcritical measurements experiment using G4-STORK and to compare the results from G4-STORK code to experimental results, as well as to results computed from MCNP computer code.

This chapter briefly describes the major modifications which were made to the GEANT4 classes and is not intended to give a complete description of the calculation methods used in G4-STORK. For a detailed methodology description it is best to refer to dissertation thesis by Liam Russell [6]. The next part of this chapter concentrates on describing the ZED-2 experiment and on modeling this experiment in G4-STORK and MCNP. However, the emphasis of this section is placed on the modeling the ZED-2 subcritical measurements experiment in the G4-STORK, as the use of G4-STORK for this particular experiment is the focus of this study.

5.1 G4-STORK

The G4-STORK code has features which make it a practical code for nuclear reactor physics calculations, especially for perturbations via the reactor core (i.e., caused by rod withdrawal, temperature oscillations and transient behavior such as non-critical state of the nuclear reactor).

The G4STORK code was developed at McMaster University. The development of G4-STORK is still in its early stage. G4-STORK is written based on the open-source Monte Carlo particle physics simulation Geant4 toolkit, as was discussed in chapter 3 (3.2.2). This code is mainly developed in a LINUX based operating system and has been neither compiled nor tested in Windows operating systems. Figure 5.1 illustrates a flowchart of the main process in G4-STORK. The program's main processes start from pre-processing the user input, and the world geometry processes the cross section data library for the world's material. Then this information is passed on to the main calculation action processes (i.e., initial and final action as defined in 3.2.3). The final results from the run action are collected and outputted into some files. G4-STORK has the capability to run in the parallel mode. The parallelization of G4STORK uses the event-level parallelism which helps to speed up the computation of the events [27]. The parallelization was implemented using TOP-C and marshalgen[1] [6]. The event-level parallelism uses a master-slave topology. There are a number of slaves that each process a number of events. Then the master is responsible for coordinating the slaves and also all the run-level results (i.e., calculating the physical parameters at the end of each run) [6].

The special feature of G4-STORK is the ability to track the evolution of the neutron population in time, including delayed neutrons. It can also model the resulting changes

[1] A full descriptions of TOP-C and Marshalgen tools can be found in the references [27] and [6]

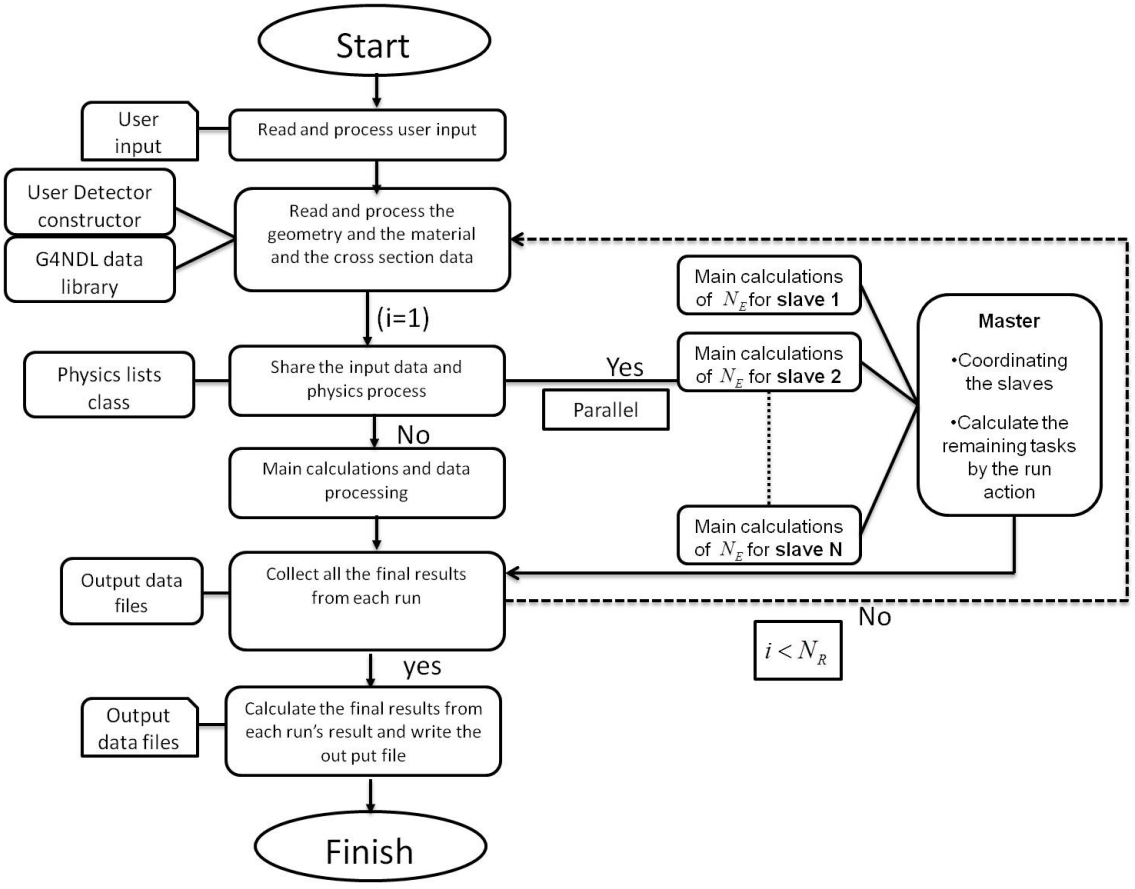


FIGURE 5.1: Data flow and main processes in the G4STORK code.

in material and geometric properties of a reactor. The time-dependent feature in G4-STORK makes this reactor physics code a good candidate for transient calculations in a nuclear reactor.

5.1.1 Data Processing

Every single volume in the G4STORK simulation world is defined as a neutron sensitive detector. These neutron sensitive detectors must collect and save the following data:

1. Check if the particle is a neutron otherwise kills the particle
2. Record the neutron survivors and delayed neutrons
3. Record the number of neutron produced and lost

4. Record the total neutrons (lost) lifetime

5. Record the positions of all the fissions

Then the saved data (by a class called TallyHit) from the sensitive detectors will be transferred to event action. The responsibility of event action is to repackage the data from TallyHit into EventData and to hold the information from TallyHit of that specific event (an integer that classifies each event fro Run Manager). After all events in a run are done, all the data will be passed to run action by run manager for a set of analysis which needs to be done before end of the run.

The run action is responsible to calculate some physical quantities, such as the run multiplication constant k_{run}(5.1), the average neutron life time, the neutron multiplication constant k_{eff}(5.2), and run duration. Then all of this information will be recorded in an output log or into the screen (C). The run action is also responsible to save the survivors and delayed neutrons into the separate files with their event identifiers before a run is over.

Run manager is a class which has many key responsibilities throughout the simulation. It is responsible for the communicating between the successive runs, getting the primaries from each event from the primary generator, passing the data from each event to the event action. Also, this class needs to update the simulation time, check for the convergence in Shannon entropy, keep the physical quantities after Shannon entropy conversion, and update the survivors' list as well as delayed neutrons' lists. The following diagram 5.2 summarizes the data processing flow in G4-STORK in addition to their responsibilities.

5.2 Implementation of G4-STORK

As was described in section 3.2.2, the GEANT4 toolkit contains libraries of functions and classes that provide the necessary information to build simulations in various fields

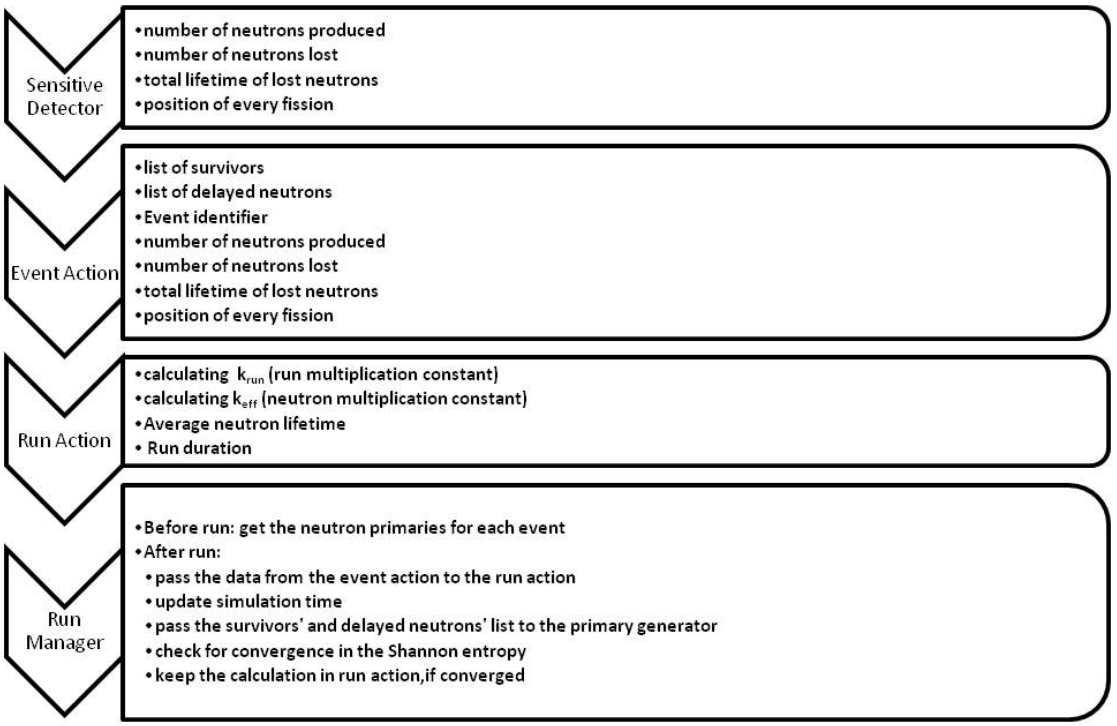


FIGURE 5.2: G4-STORK data processing hierarchy.

in science. Even though this versatile software has many applications, it has little applications in the field of nuclear engineering. Therefore, to use Geant4 for nuclear reactor simulations, some of the classes and functions had to be modified [6].

The G4-STORK code was developed by using the modified classes and functions in GEANT4 for creating a reactor physics code to simulate the evolution of a neutron population over a period of time in any medium [6]. To simulate the true neutron evolution at a regular time interval, the evolution of the neutron population should not be disrupted. The following subsections describe the development of the G4-STORK code to achieve this goal and highlight the details which are more important to this project.

5.2.1 Neutron Population Stabilization

As discussed in section 2.1.2,the multiplication factor (k_{eff}) is a quantity that determines the change of the neutron population of a nuclear reactor in time. The neutron population changes in time follow an exponential law given by equation 2.11. Figure 2.9

illustrates the three cases of the criticality when the neutron population is constant in time or it is critical 2.13, and the non-critical cases when the neutron population is not constant in time subcritical 2.14 or supercritical 2.12.

Thus, simulating non-critical systems depend upon having a fixed number of neutrons to avoid losing too many neutrons at subcritical cases or of producing a large number of neutrons in supercritical cases. Since G4-STORK is a time-dependent code, the number of neutrons is kept constant by renormalizing the number of neutrons at fixed time intervals. Each of these time intervals, which are the period between renormalizations, is called a **run**.

5.2.1.1 Renormalization Method

At the beginning of the simulation in G4-STORK, a set of primary neutrons is introduced to the system (i.e., at time t_0). All of the neutrons are tracked throughout the run, and are stopped by reaching the time $t_0 + T$, which is considered as the end of run. At the beginning of the next run, the survivor neutrons from the previous run are renormalized to the initial number of primary neutrons at the beginning of the previous run. The renormalization of the neutron population occurs through either deleting or duplicating of individual neutrons by creating a list of targets that are uniformly distributed across the list of survivors [6].

These periodic renormalizations of the neutron population are to ensure that there are a sufficient number of neutrons for an accurate representation of the neutron's spatial and energy distribution of the non-critical systems. Moreover, selecting targets uniformly across the survivor list does not affect the spatial and energy distribution of the neutron population [6].

Figure (5.3) illustrates renormalization process for two consecutive runs. The run i contains four events, and every event starts with N primary neutrons.

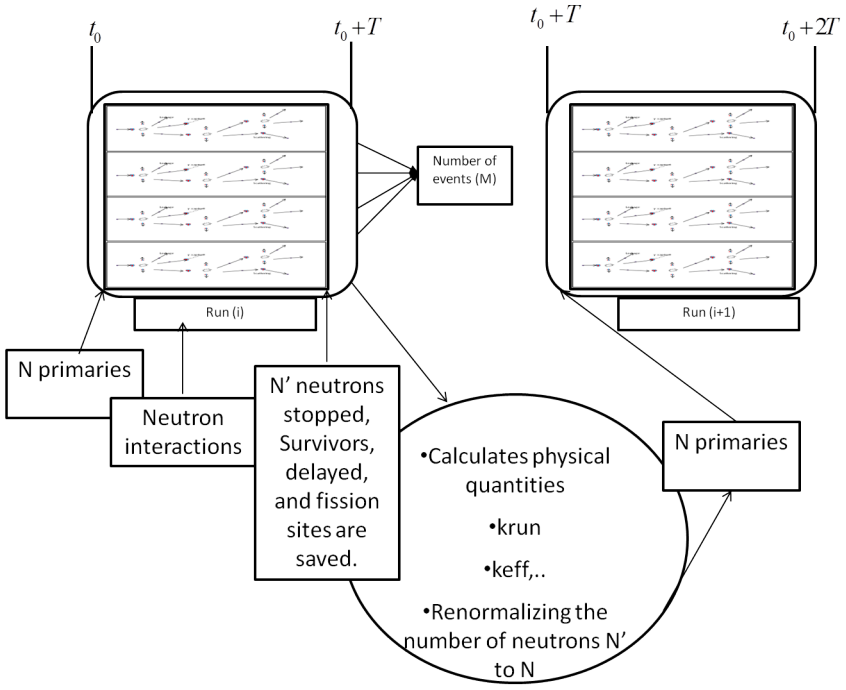


FIGURE 5.3: A schematic of G4-STORK process between the runs.

The N neutrons are tracked as they go through the interactions with the method that was described in section 3.2.3. All the neutrons are eventually stopped at time $t_0 + T$ or at the end of the run. At the end of each run, the survivor neutrons, along with fission sites and delayed neutrons, are saved in separate files. Also some physical quantities, such as k_{run} and k_{eff}, are calculated before the run is finished. The k_{run} which is called the **run multiplication constant** quantifies the absolute change in the number of neutrons in a time interval $[t_0, t_0 + T]$ in run (i) and is defined as:

$$k_{run} \equiv \frac{N(t_0 + T)}{N(t_0)} \tag{5.1}$$

k_{run} is an important quantity for renormalizing the neutron population for the next run *i+1*. The neutron population (N') is renormalized to the number of initial primary neutrons (N) from the selected survivors by either duplicating or deleting depending on k_{run} before the next run starts, which is summarized as follow :

- $k_{\text{run}} = 1$, the neutron survivor list is used as the primary neutron population for the next run

- $k_{\text{run}} < 1$, some of the neutrons are duplicated from the previous run neutron survivor list
- $k_{\text{run}} > 1$, some of the neutrons are deleted from the previous run neutron survivor list

5.2.2 Computed Quantities

Every simulation in G4-STORK is made of several of these temporal run divisions. At the end of each run, before the neutron population is renormalized, some of the reactor's physics characteristic quantities, such as the number of neutrons lost and produced, are extracted from the run outcome [2]. k_{eff} for an instance in time t is defined as equation 2.17. On the other hand, calculating k_{eff} stochastically requires a time interval in which the number of neutrons produced and lost are tallied.

G4-STORK uses the dynamic method that was discussed in section 3.2.1 for the criticality calculations. This method computes k_{eff} value using 3.20, which is a ratio of the neutron productions over the neutron loss in a time interval T of run *m*. Thus, equation 3.20 is rewritten as:

$$k_{\text{eff}} = \frac{N_P(m)}{N_L(m)} \tag{5.2}$$

Where N_P and N_L are number of neutron produced and lost in run *m*, respectively.

5.2.3 Delayed Neutrons

The delayed neutrons are not produced in the current run unless they either come from a previous delayed neutron list or from being generated instantaneously. As was mentioned earlier, the delayed neutrons of the current run are saved in a separate file for the future runs (as shown in Figure 5.3). At the start of each run, the delayed neutron list is

[2] An example of the log output files is shown in appendix C

checked to see if any delayed neutrons should be born in the next upcoming run. In this case, the delayed neutrons are removed from the delayed list and added to the survivor list before the renormalization processes occur.

G4-STORK generates the delayed neutrons in three ways:

First is production of the delayed neutrons to be ignored. This method is advantageous for the systems in which the delayed neutrons are not important.

Second method is to generate the delayed neutrons promptly at the time of the fission, but the initial momentum of the delayed neutrons is still sampled from a delayed neutron data. This method is only useful for near critical systems.

Third option is to produce the delayed neutrons in the time that they would naturally decay from the precursors. Thus, the concentration of each precursor group is calculated and saved from an older steady state of the system. In reality this list of the delayed neutron precursors is a list of coordinates and the decay times at which a delayed neutron will decay in future. Thus, at the beginning of every run, the delayed neutron precursors need to be sampled, checking if any precursor would decay (produce a delayed neutron) in the current run. Also, the precursor tally for each group has to be renormalized by the same weighting factor that is used to renormalize the survivors ($N(i+1)$) at the beginning of each run

$$C_l(i+1) = C_l(i) \times \frac{1}{k_{\text{run}}(i)} \tag{5.3}$$

where l represents the l^{th} precursor group and C_l is the precursor concentration of group l.

However, the last method is still being developed further, and this option was not used for this project.

5.2.4 Boundary Condition Options

The boundary conditions define what happens to the neutrons that end up in a region outside of the world volume. The boundary condition features are especially important for reactor geometry simulation using infinite lattice simulation, and core's sector simulation (i.e., quarter core, half core, etc.). Since Geant4 does not have such boundary conditions, they were developed in G4-STORK [7]. There are three options available in G4-STORK:

1. Black boundary: the neutrons that cross this boundary are killed.
2. Reflective boundary: the neutrons that hit the boundary are reflected back into the geometry.
3. Periodic boundary: the neutrons that hit the boundary are removed from the geometry and are reentered from the opposite side of geometry.

The periodic and reflective boundary conditions in G4-STORK were implemented as step-limiting process. For the periodic boundary, the step-limiting process acts on those neutrons which reach the boundary of outermost geometry in the simulation by killing the neutron and reentering an identical neutron (a neutron with the same property) from the opposite side of the boundary of the geometry [7].

For the reflective boundary, the neutrons that reach the boundary are again killed and the same-property neutron is reborn from the same boundary in geometry but in an opposite direction of its previous movement.

A useful feature in the G4-STORK simulation is the capability of simulating different boundary conditions simultaneously. For instance, G4-STORK can simulate a square box with four of its sides set as reflective and the rest of them as black boundary conditions. This feature is useful for the sector of a reactor core calculation, such as a quarter-core having two reflective boundary sides that represent the other three quarter of the core and a black boundary which represents the graphite wall boundary condition.

5.3 Modeling the ZED-2 Subcritical Reactivity Measurements Experiment in G4-STORK

The purpose of this study was to model the subcritical measurements experiment in ZED-2 using the G4-STORK code. Since G4-STORK has the ability to simulate the transient behavior such as subcritical calculations, it makes the prediction of such conditions in a reactor more plausible to calculate. Therefore, the simulation of this experiment was implemented in G4-STORK and was compared to the MCNP code for the same experiment.

This section describes the ZED-2 experiment briefly. More detailed information can be found in the published paper [4]. Modeling this experiment in G4-STORK is then discussed in detail followed by a brief discussion of modeling this experiment in MCNP. It must be mentioned that the author has made no contribution to the development of the full core MCNP model for this particular experiment and that the input file was provided to the author by ZED-2 group at AECL (now CNL).

5.3.1 The ZED-2 Subcritical Experiment

The study of subcritical states is not only important for monitoring the approach to the critical state but has many significant applications such as preventing costly down time of power reactors and problems due to monitoring the lower signal to noise ratio from instruments [4]. One of the more fundamental uses of subcritical calculations is using them as benchmark measurements. However, not many subcritical experimental data are available to validate the results of such subcritical calculations.

The following subsections describe the ZED-2 reactor as well as describing the experiment in more details. This section is a brief description of the ZED-2 reactor and the

configuration that was used in this experiment, the reactivity measurements and calculations, and the modified MCNP code which was used for this study.

5.3.1.1 ZED-2 Reactor

ZED-2 reactor is a low-power ($\sim$ 5-10 W), heavy-water moderated, versatile tank-type research reactor. The reactivity of the reactor is controlled by manually adjusting the moderator height.

The ZED-2 geometry for the sub-criticality measurements used the CANFLEX-LEU type fuel in a 52-assembly square lattice. Each fuel assembly consisted of five fuel bundles that each have a height of about 50 cm, and no coolant was included in the fuel assemblies for this experiment. The Figures 5.4 and 5.5 show a top view and a side view of the reactor core and a top view of the lattice cell, respectively. These Figures 5.4 and 5.5 are generated by the GEANT4 visualization.

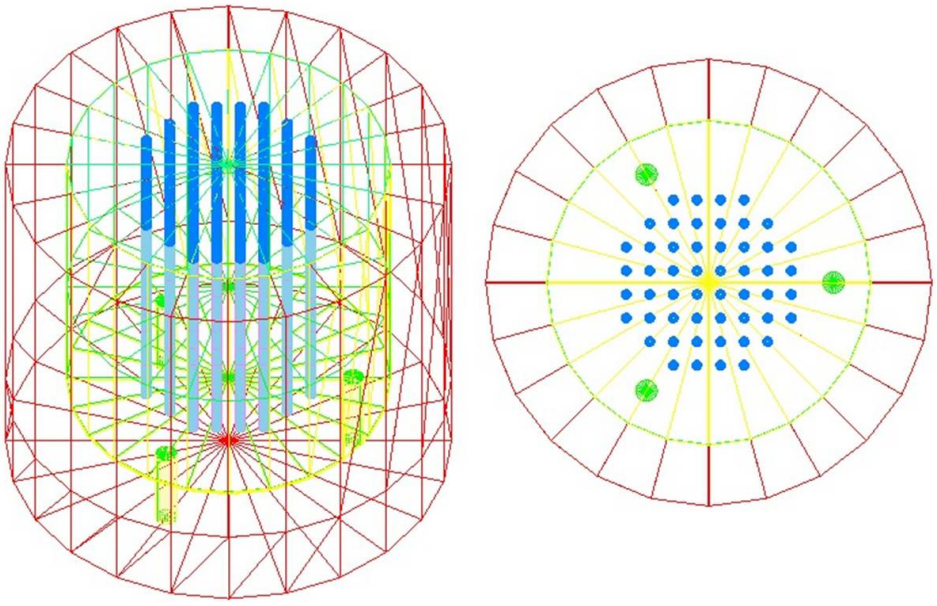


FIGURE 5.4: Angle view (left) and cross-sectional view (right) of the ZED-2 reactor.

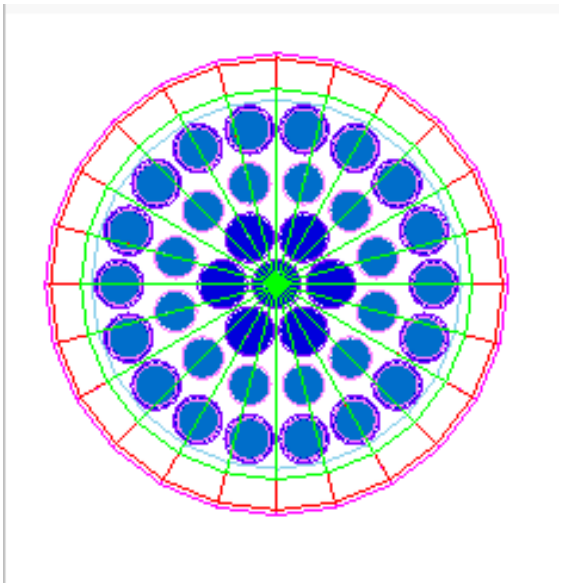


FIGURE 5.5: Cross-sectional view of CANFLEX fuel bundle.

5.3.1.2 Reactivity Measurements

The reactor was operated at about 100 W for one hour for each subcritical height (draining between 5 to 25 cm of heavy water from the reactor tank). The counts were collected using a fission chamber. The fission chamber is located in the D_2O reflector [4]. The results from the measurements were used to determine the relationship between moderator height and the inverse of the count rate as shown in Figure 5.6 [4].

The subcriticality values were calculated using the inverse point kinetics method. This method is described briefly below.

5.3.1.3 Inverse Point kinetics Method

Solving the point kinetics equation 3.10 for ρ gives the following:

$$\rho = \beta + \frac{dn}{dt}\frac{\Lambda}{n(t)} - \frac{\Lambda}{n(t)}\sum_{k=1}^{k_{max}} \lambda_k C_k(t) + \frac{S_0}{n(t)}. \tag{5.4}$$

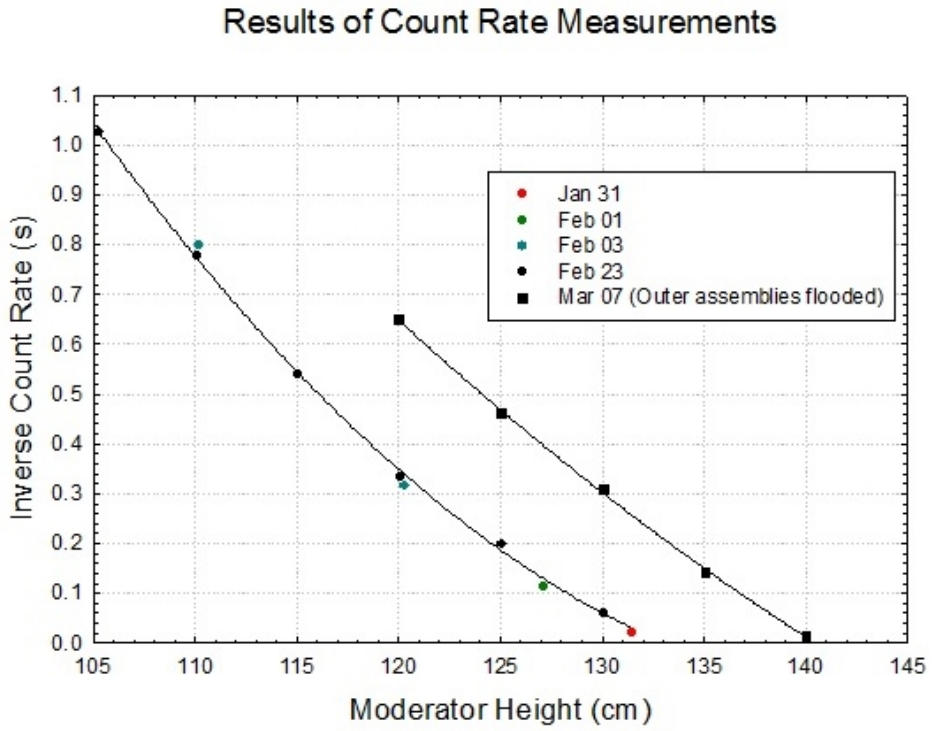


FIGURE 5.6: Inverse subcritical count rates as a function of the moderator height [4].

Then solving the other point kinetics equation3.11 for the precursor concentration gives:

$$C_k(t) = \frac{\beta_k}{\Lambda} \int_{-\infty}^{t} n(t) e^{-\lambda_k (t-t')}. \tag{5.5}$$

Substituting 5.5 to 5.4 is:

$$\rho = \beta + \frac{dn}{dt}\frac{\Lambda}{n(t)} - \frac{1}{n(t)} \sum_{k=1}^{k_{max}} \lambda_k \beta_k \int_{-\infty}^{t} n(t) e^{-\lambda_k (t-t')} + \frac{S_0}{n(t)} \tag{5.6}$$

The equation 5.6 is called inverse point kinetics [2] and was used to calculate the reactivity for the subcritical experiment by using the following parameters in the Table 5.1.

The sub-critical state of the reactor was achieved by the step-wise draining of the moderator. The steps of draining the moderator were from about 5 to 25 cm out of the reactor vessel for about one hour. These measurements taken during the transient were

β	Sum of all the delayed fractions $\sum_{i=1}^{i_{max}} \beta_1$
λ_k	The decay constant; Its values are known depending on the fuel material
Λ	The mean generation; Its value was evaluated by some MCNP calculations
$n(t)$	the neutron number density that comes from the count rates of the fission chamber
S_0	the external source which is usually ignored

TABLE 5.1: Variables used in the calculation of subcritcal reactivity using equation 5.6

used to determine the relation between k_{eff} and moderator height. Figure 5.7 shows the moderator height versus k_{eff} [4]. Figure 5.7 shows two sets of experiments:

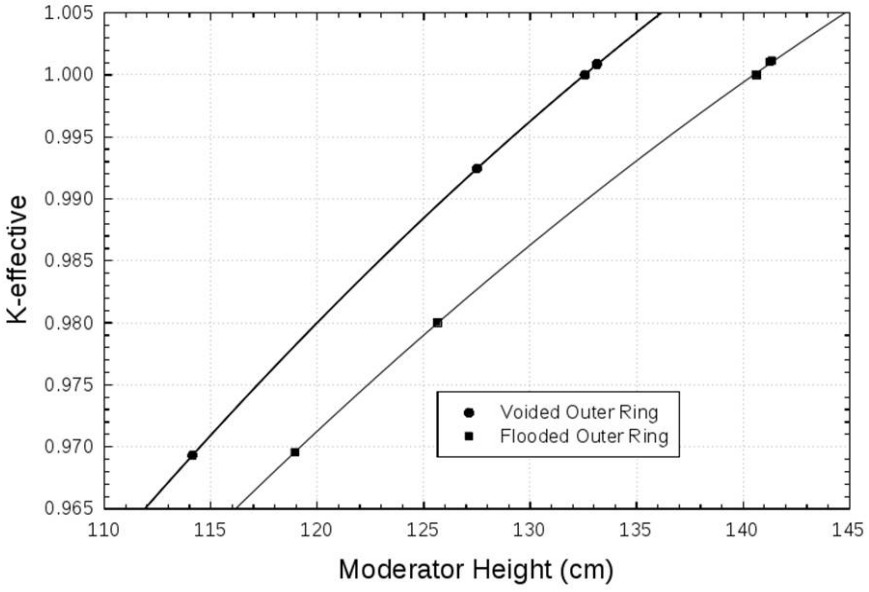


FIGURE 5.7: Subcriticality measurements as a function of the moderator height [4].

- Using only air-cooled fuel assemblies (voided) in the core lattice
- Using light-water-cooled fuel assemblies only for the outer ring of the core lattice (flooded)

In this work, however, the results for the voided experiment are presented.

5.3.1.4 MCNP Code Calculation

The experimental results were intended to be compared to MCNP calculations for k_{eff} calculations. However, some modifications on MCNP k_{eff} calculations were required, since the MCNP only computes the k_{eff} near critical states. Therefore, Atfield et.al. used the following modification for criticality calculations in MCNP: The KCODE card calculation in MCNP is responsible for tallying the criticality calculations data. For these calculations, all the neutrons are tracked through a series of fission cycles. The source for each cycle is determined by the fission sites of the previous cycle [4]. However, all the KCODE tallies are collected at the steady state, and KCODE for subcritical systems do not include any multiplication factors [17]. The MCNP manual suggests that the subcriticality calculations can be evaluated by multiplying the results by the system multiplication [17]. Atfield et.al. modified the criticality calculations in MCNP by introducing *M* defined as: the multiplication factor in the KCODE card calculation as [4]

$$M = 1 + \sum_{i=1}^{\infty} \Pi_{j=1}^{i} k_j. \tag{5.7}$$

Then k_{eff} is

$$k_{\text{eff}} = 1 - \frac{1}{M}. \tag{5.8}$$

Since this paper [4] only shows the preliminary results of the subcritical measurements experiment, the results from the MCNP criticality calculations were not available in Atfield et.al. [4]. However, the MCNP input file for this particular experiment was provided to us for this study. For this reason, the MCNP criticality calculations are also used to be compared to the G4-STORK.

5.3.2 G4-STORK Implementation of the ZED-2 Subcritical Experiment

The purpose of the G4-STORK modeling needs to be clearly defined before proceeding further in this section. The objective of using G4-STORK is that this code has the ability to model transient changes in both material and geometric properties. This ability makes G4-STORK a very good candidate to model the transient behavior of the subcritical experiment.

5.3.2.1 Geometry Setup

One of the great advantages of using G4-STORK is that it has the ability to model the complex, heterogeneous 3-Dimensional geometries at the level of realistic neutron interactions in a reasonable time. The G4-STORK code is versatile to handle from a simple geometry with one kind of material (such as a solid sphere) to a complex geometry (such as a reactor).

All the simulation world types (i.e., a solid sphere or ZED-2 reactor) are built using a class called "StorkWorld[3]". This class creates the simulation geometry and the materials based on the user input file.

For this study, two simulation worlds were added to the G4-STORK code: A full core of ZED-2 reactor and a quarter of the core of this reactor. The world constructors for both geometries (i.e. full and a quarter core) are shown in the Appendices A and B.

5.3.2.2 Full Core

All the physical dimensions of ZED-2 reactor were taken from a ZED-2 report listed in the reference [5]. The simplified model of the ZED-2 geometry was modeled in

[3] All the G4-STORK classes are defined in the dissertation thesis by Liam Russell [6]

G4-STORK; this simplified model was defined in the report [5], and shown in Figure 5.8. The parts of the reactor's geometry which have a contribution less than a few

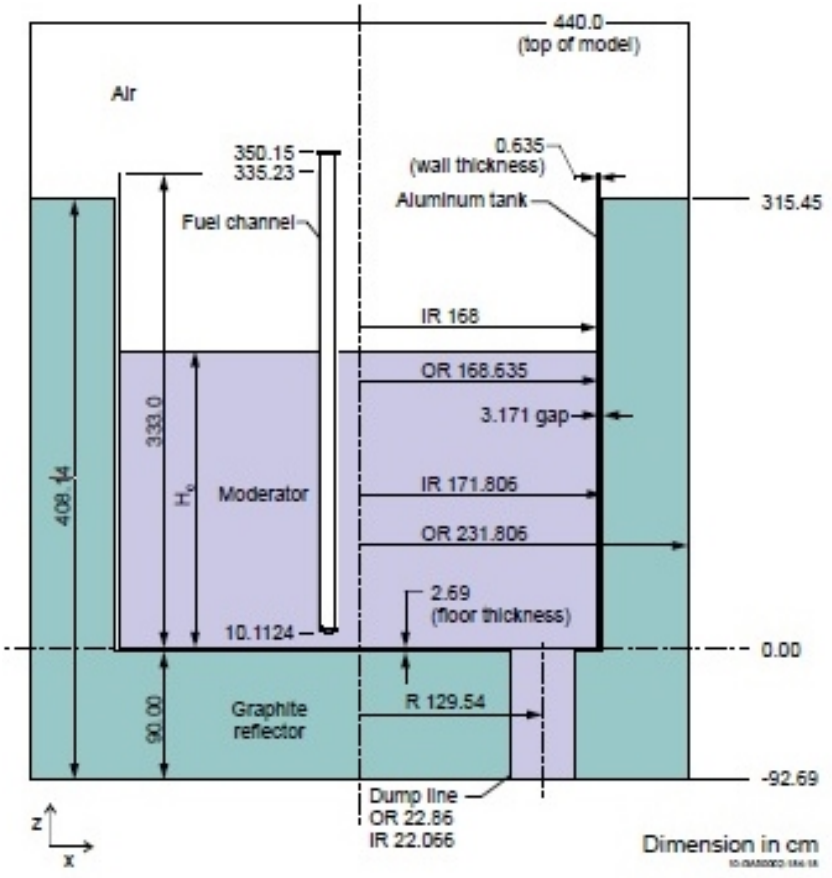


FIGURE 5.8: The simplified model of the ZED-2 reactor [5]

hundredth of mk were ignored in the G4STORK model. The geometry shown in Figure 5.4 illustrates the full core geometry for the full core simulations. The lines in Figure 5.4 are representative of the solids or the volumes, and they appear if the visualization attributes in Geant4 are not set to solid.

5.3.2.3 Quarter Core

The quarter core was modeled in the G4-STORK code for two reasons. First, it can save computational time because of the symmetry. The second reason was to estimate the neutronics calculations with a better statistics.

The boundary conditions are then defined as reflective boundaries in the sides ending by the moderator and the graphite reflector side of the reactor has the normal boundary condition. Having the boundary conditions as stated makes the quarter core a representative model for the full core. Therefore, the results from the quarter core can then be extended for the full core. The multiple boundary conditions (mix of reflective and regular) were added to the G4-STORK code for this study. Figure 5.9 shows the radial cross section of the quarter core generated in the G4-STORK code.

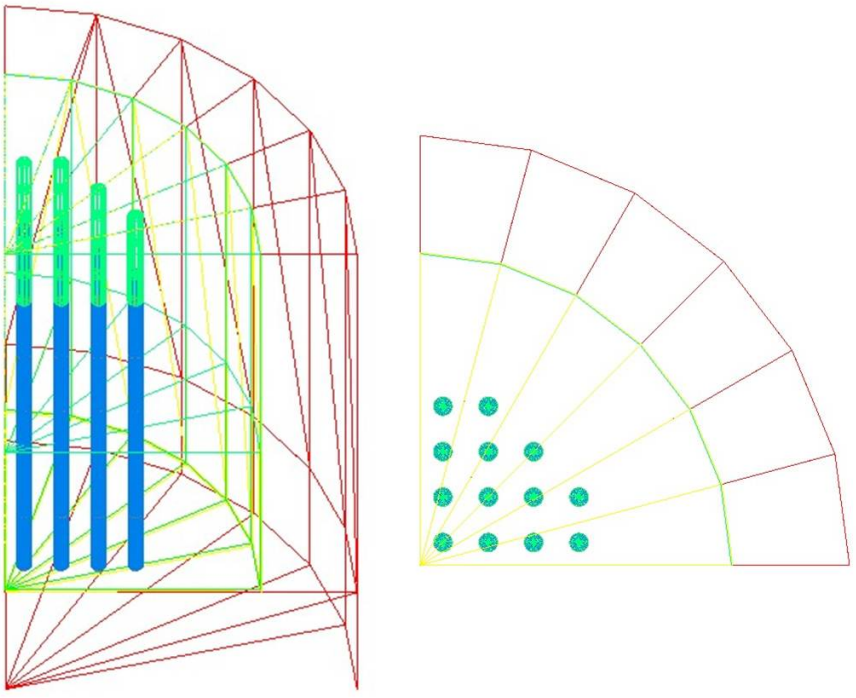


FIGURE 5.9: Angle view (left) and cross-sectional view (right) of the quarter ZED-2 reactor core.

5.3.2.4 Material

The materials that are defined in the G4-STORK code are identical to the materials used in the MCNP model for the comparison purposes. In general, the materials are made of elements that are made of isotopes. The materials description in G4-STORK follow the same trend and are described by three classes. The descriptions of these classes are illustrated in the flowchart below (5.10). Since all of the materials description in MCNP

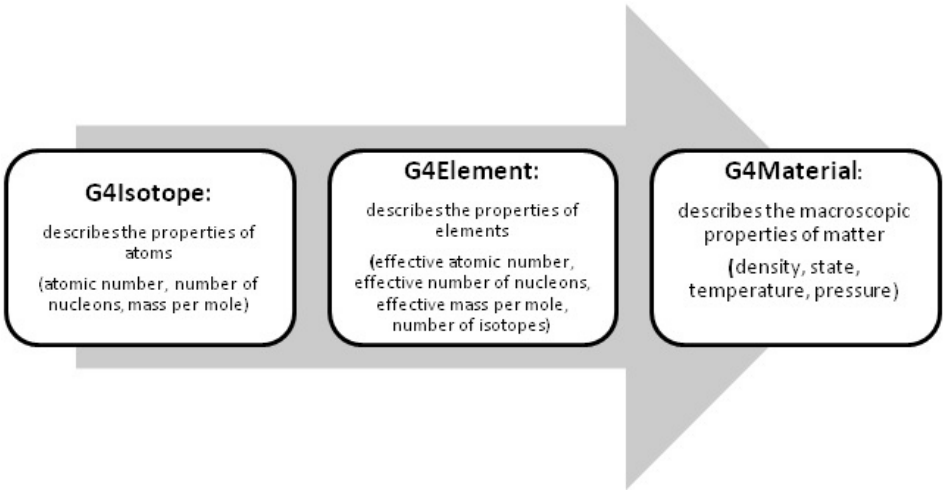


FIGURE 5.10: Material description flowchart.

input were not defined in the appropriate form for G4-STORK, some extra calculations to convert the weight fraction of some isotopes to their atomic weight percent was required. As mentioned earlier, the materials are defined in the constructor world and can be seen in either the full core or a quarter core classes A and B.

5.3.2.5 Cross section Data Library

G4-STORK uses the G4NDL data library. The cross section data in G4NDL are collected from a number of evaluated data libraries such as BROND-2.2, CENDL-31, ENDF-B/VI.8, ENDF-B/VII.0, JEFF-3.0, JEFF-3.1, JENDL-3.3, JENDL-4.0, and are reformatted to the G4NDL format [28]. The cross sections in G4NDL are point-wise cross sections for accuracy purposes.

G4NDL contains the neutron interaction data for neutron energies ranging from 0 to 20 MeV [29]. Every neutron interaction (e.g. elastic scattering) is divided into interaction cross sections and final states data [4] [6].

The cross section data in G4NDL are zero Kelvin data[28]. The cross section temperatures that are passed on by the detector description are used to Doppler-broaden the cross sections on the fly[28]. On-the-fly Doppler broadening in GEANT4 is done with

[4]The state of the secondary particles

assumption that the motion of the nuclei in the material follows the free-gas model and using a Monte Carlo integration technique to Doppler-broaden the temperature to the described temperature [28]. The general equation used to calculate the cross section data on the fly is given by[30]:

$$V_n\sigma(V_n,T) = \int_{\text{all V\_t}} V_r\sigma(V_r,0)P(V_t)dV_t \tag{5.9}$$

where all the variables in above equation 5.9 are defined in Table 5.2

V_n	The incoming neutron velocity
σ	The cross section at temperature T
T	The local temperature of the material
V_t	The target velocity
V_r	The relative velocity ($\mid V_n - V_t \mid$)
$P(V_t)$	The normalized Maxwell-Boltzmann distribution

TABLE 5.2: Variables used in equation 5.9

The on-the-fly method that is used in Geant4 HP models is as described below [6]:

On-the-fly Doppler broadening method in Geant4

- **While** the difference of $\bar{\sigma}$ before and after its update is $< 3\%$
- **For** i=1 to 10 **do**
  - Sample V_t
  - Calculate cross section $\bar{\sigma}$ by equation 5.9
- **End For**
- Update $\bar{\sigma}$
- **End While**
- Return $\bar{\sigma}$

The technique used to Doppler broaden the cross section on the fly samples the velocity of the nucleus and determines the neutron interaction cross section in the rest frame of the sampled nucleus. This process described above carries on until the average cross section value converges within the given limit (%3). Concerning the accuracy, 3% error is large. Although the averaged error in a large number simulation can be smaller (i.e., increasing the number i). Thus, this technique can be computationally expensive for simulating geometries such as ZED-2 reactor. Also, on-the-fly Doppler broadening technique becomes increasingly expensive as the temperature of the material increases [6].

Chapter 6

Results and Discussion

This chapter discusses and analyzes the computer simulated results for the subcritical reactivity measurements in ZED-2 by G4-STORK; as well, it compares the G4-STORK results with MCNP and the experimental measurements for this experiment. The G4-STORK and MCNP models were described in details in sections 5.3.2 and 5.3.1.4, respectively. Also the experiment measurements was discussed in section 5.3.1.

In the following sections, the results are presented for ZED-2 full core and quarter core calculations of this ZED-2 experiment. First, the G4-STORK full core results are elaborated on and are correlated to the MCNP calculations, as well as to experimental measurements of this ZED-2 experiment. Later, the ZED-2 quarter core calculations by the G4STORK are analyzed and compared with the ZED-2 full core results.

6.1 ZED-2 Full Core

The following G4-STORK results were obtained using 9×10^5 primary neutrons per run. Each simulation lasted for 200 runs, and every run had a period of 1 ms. Each simulation was initialized from a point source at a fuel location. The choice of the point

source at a fuel pin was to achieve convergence in space and energy for the neutron distribution to the fundamental distributions implied by the world definition.

The primary neutron energies are sampled from a Gaussian distribution centered at 2 MeV. The delayed neutrons were generated using the G4-STORK second method 5.2.3 by producing them instantaneously at the time of the fission for these simulations. But their characteristics are sampled from the delayed neutrons distributions for energy, momentum, and the life time. G4-STORK uses G4NDL cross section libraries that are based on ENDF/B-VII, JEFF-3.1, JENDL-4.0 cross section data.

The criticality and subcriticality measurements for ZED-2 were computed in G4-STORK using the simulation setup described above. Three simulations were conducted for critical (132.7 cm) and two sub-critical (127.7 cm and 114.0 cm) moderator heights. Thus, all of the following analysis is done for all of these three heights. Figure 6.1 presents the k_{eff} values in time on left and k_{run} values in time on right. Each data point represents the result of a single run. It can be seen in Figure 6.1 that the estimated k_{eff} and k_{run} values fluctuates a lot for the first few runs. However, after the initial convergence period the simulated results seem to be stabilized around a constant average value. Thus, the results from the first 50 runs were discarded.

[htb]

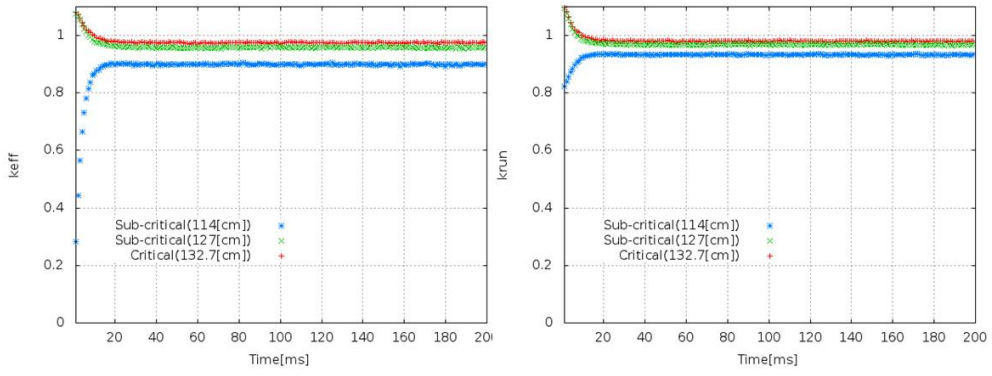


FIGURE 6.1: The effective (on left) and the run (on right) multiplication factors in time.

As was mentioned earlier, the convergence in the multiplication factors results from the convergence of the neutron spatial distribution and energy spectrum in time. Figures 6.2

and 6.3 show the spatial distributions and the energy after the convergence was attained for the critical height. The top part of Figure 6.2 and Figure 6.3 present the neutron distributions in time 50 ms and 200 ms.

At these times, both the neutron distribution and energy had been evolved and stabilized in time. In the next section 6.2, the stabilization of the neutrons in time is demonstrated for a ZED-2 quarter. The bottom part of Figure 6.2 illustrates a schematic of the ZED-2 reactor, including all dimensions in the geometry for a better understanding of the graph[1].

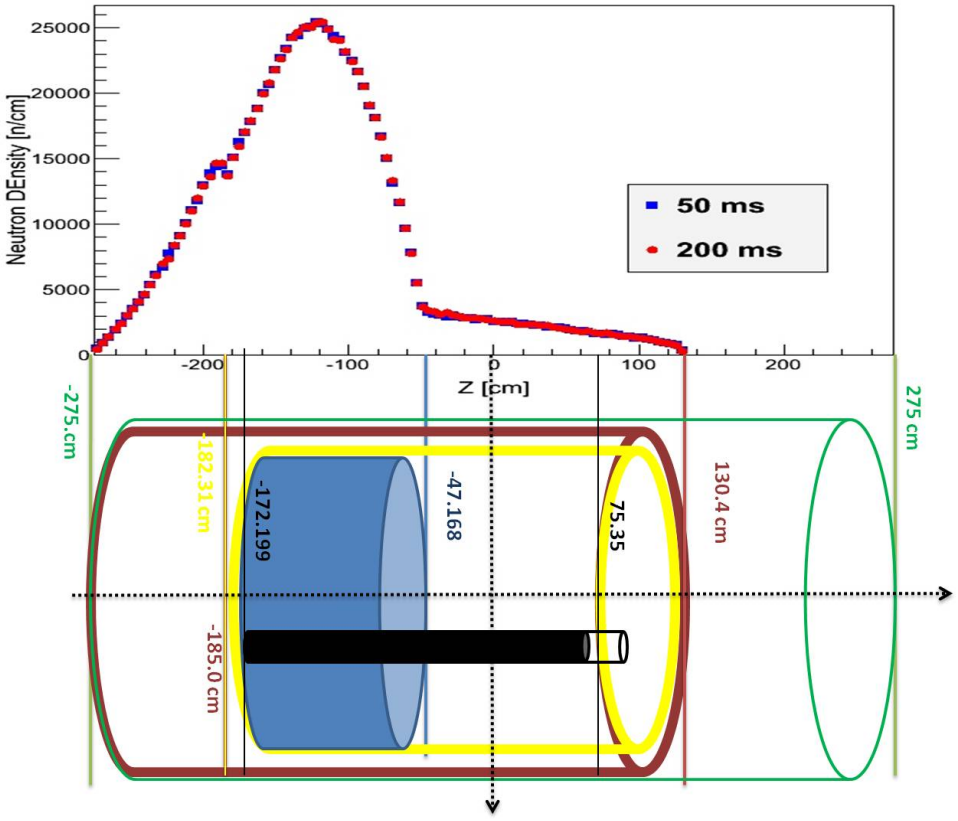


FIGURE 6.2: Converged neutron spatial distribution (top) and the schematic of ZED-2 in Z-axis including all the dimensions(bottom).

Some additional reactor physics properties were obtained by G4-STORK. Such parameters are the survivor neutrons[2] locations and their momentum at the end of each run

[1] In Geant4, all the geometries in the simulation are defined relative to the world's origin which is located at the middle of the world geometry.

[2] Survivor neutrons are the neutrons which reach the end of the run and are stopped at time $T + t_0$ (the end of the run period).

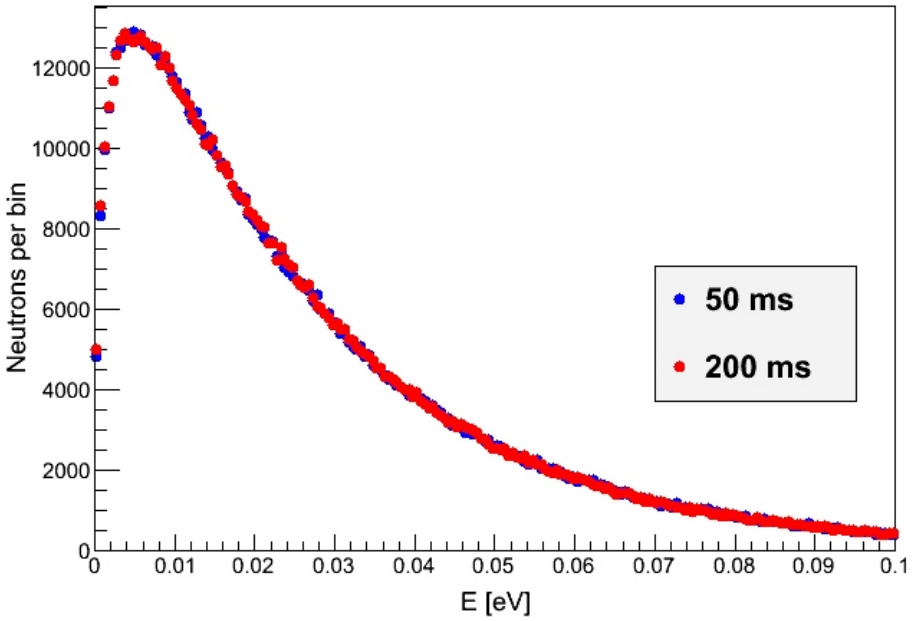


FIGURE 6.3: Converged energy spectrum at time 50 ms and 200 ms.

as well as the locations and energy of all the neutrons which were born from fission interactions [3].

The survivor neutrons from the last run and fission neutrons were collected by end of each simulation. The following graphs are plotted using the data from the critical height simulation.

The top portion of Figure 6.4 shows the neutron distributions in XY-plane (top-view) along with a schematic illustrating top-view of the ZED-2 reactor and all the dimensions in the bottom portion of this Figure. As seen in Figure 6.4, the outer blue annulus represents the graphite wall, and the dark blue color illustrates that the neutron population is relatively lower than anywhere else in the reactor. As it gets closer to the center of the reactor, the neutron distribution increases as expected. The holes in Figure 6.4 represent the fuel channels, and as expected the inner channels have more fissions and therefore more neutrons (light green holes) compared to the outer ones (light blue color). As expected, the maximum flux occurs at the center of the reactor core (as shown in Figure

[3] See appendix D for an example of survivors source file.

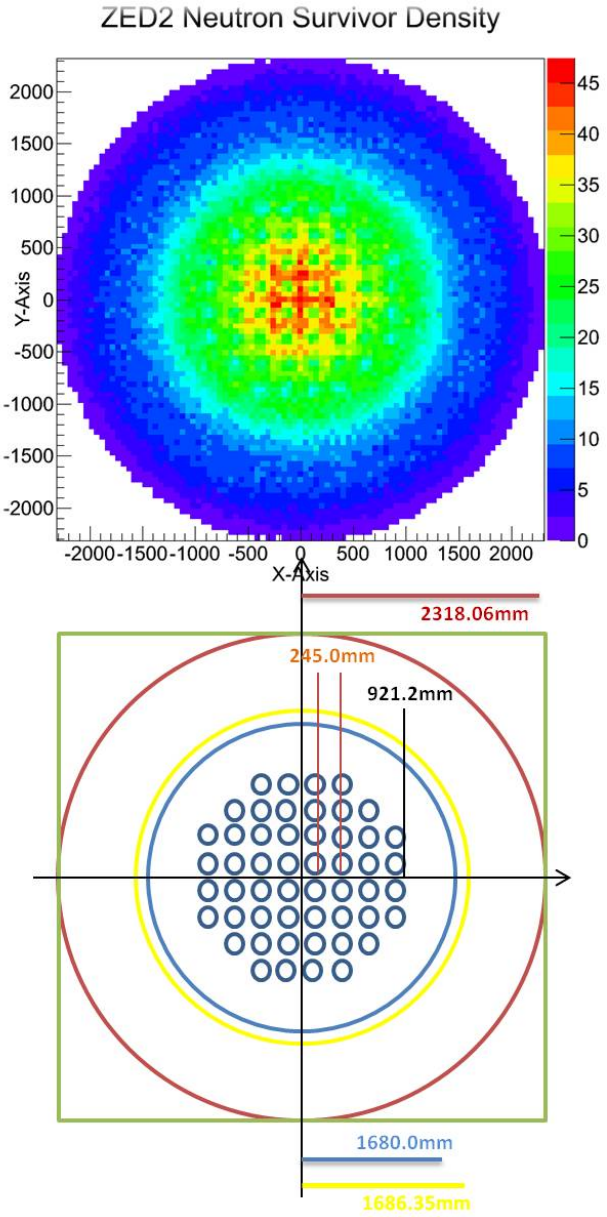


FIGURE 6.4: Neutron distribution in XY-plane (top) and ZED-2 topview schematic (bottom).

6.5) and drops approximately as a Bessel function as function of distance from the core.

Figure 6.6 illustrates the locations of all the fission events in two dimensions (XY-plane). The top part demonstrates the XY view of the whole reactor, and it is expected all the fissions interactions occurred in the fuel pins. Also the fuel channels in center of the core appear to have more fission (red colored channels) which is again anticipated. This can be better seen in the second graph in Figure 6.6. The last graph in Figure

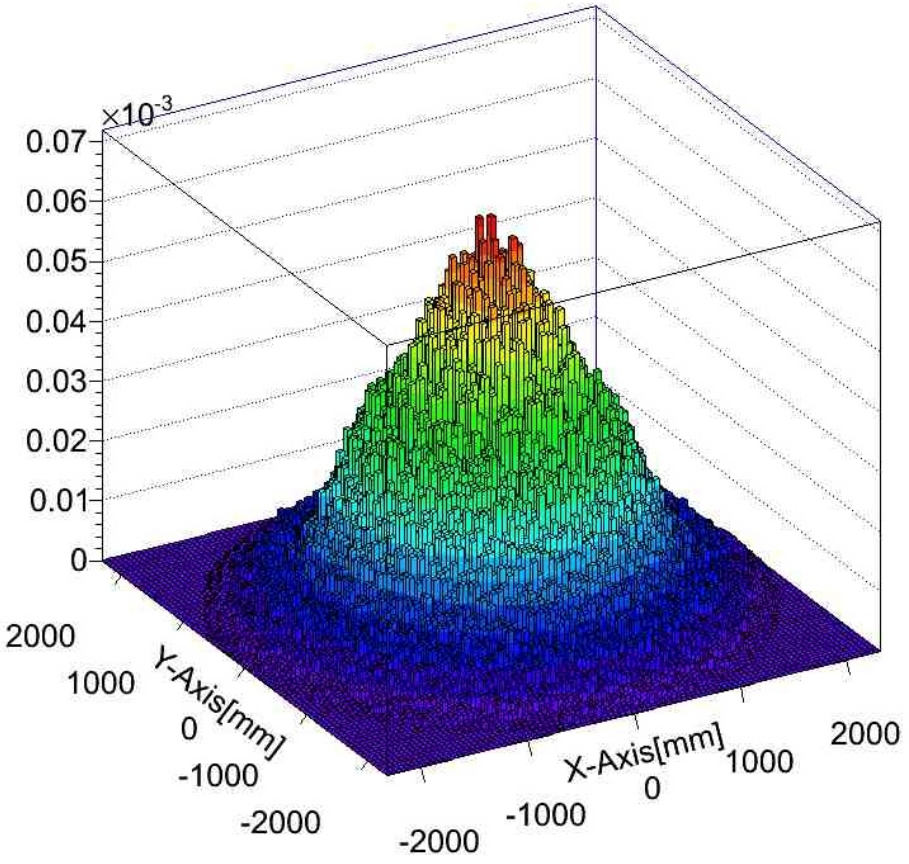


FIGURE 6.5: Neutron flux distribution plotted by 2-dimensional histogram.

6.6 indicates that the outer fuel pins contain more fission events than the inner fuel pins which demonstrate the self shielding phenomena. 6.7 shows the three-dimensional views of all the fission interactions throughout the simulation.

The simulated effective and run multiplication factors (discussed in section 5.2.3 equations 5.3 and 5.2) are shown in table 6.1. The statistical uncertainties are given as standard deviations.

Height	Average k_{eff}	Average k_{run}
Critical (132.7 cm)	0.9747 $\pm$ 0.0018	0.9799 $\pm$ 0.0014
Subcritical (127.7 cm)	0.9583 $\pm$ 0.0019	0.9683 $\pm$ 0.0014
Subcritical (114.0 cm)	0.8994$\pm$ 0.0022	0.9332 $\pm$ 0.0013

TABLE 6.1: Effective multiplication and run multiplication factors calculated using G4-STORK for ZED-2 full core

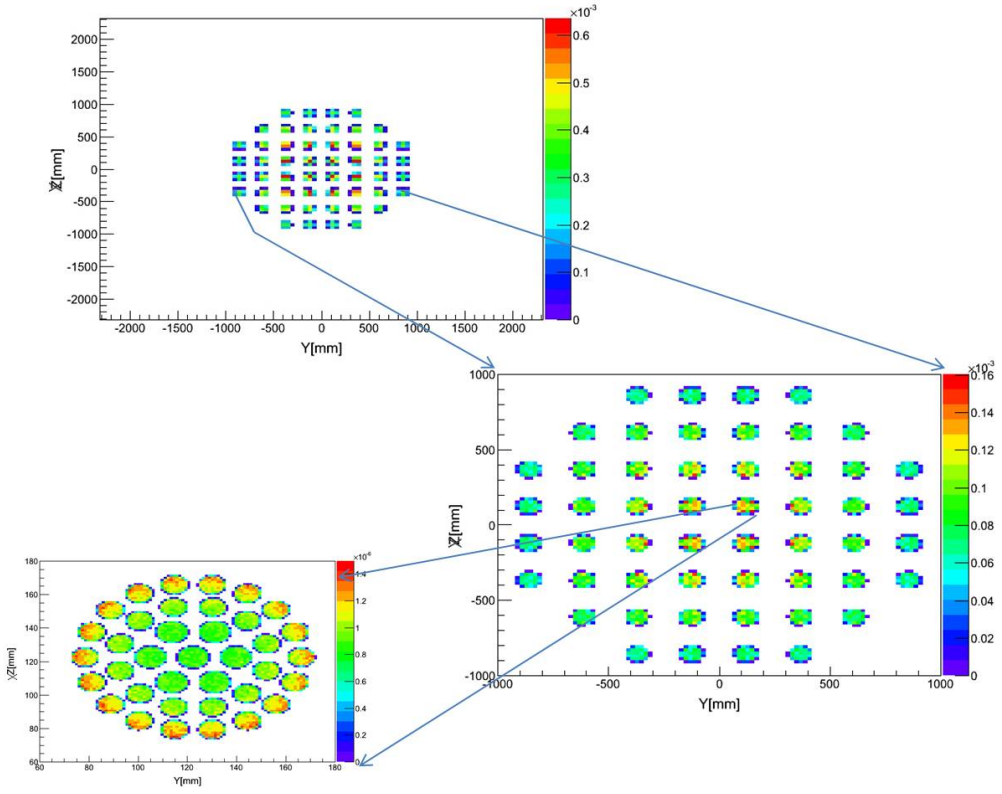


FIGURE 6.6: Fission site locations in xy plane top-view.

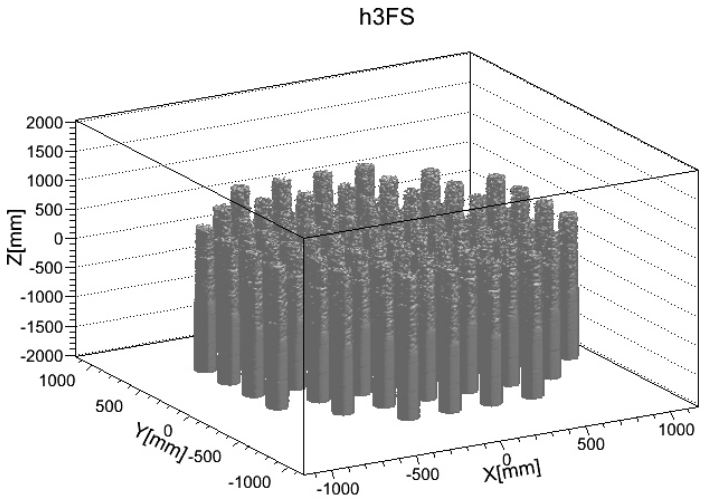


FIGURE 6.7: Three dimensional fission site locations.

The results of k_{eff} and k_{run} differ in a few mk for the critical height, and the discrepancy between the two multiplication factors increases as the moderator level decreases.

Table 6.2 shows the outcomes of the simulated multiplication factors with their uncertainties for MCNP, G4-STORK, and the experimental measurements. The MCNP

results presented in Table 6.2 are produced using 9×10^5 neutrons for 200 generations. All the MCNP simulations used ENDF/B-VII cross section data.

Critical(132.7cm)	Average k_{eff}	Average ρ_{eff}
G4STORK	0.9747 $\pm$ 0.0018	-25.96
MCNP	0.9975 $\pm$ 0.0009	-2.55
Experiment	1	0
Subcritical(127.7cm)	Average k_{eff}	Average ρ_{eff}
G4STORK	0.9583 $\pm$ 0.0019	-35.71
MCNP	0.9893 $\pm$ 0.0007	-10.8
Experiment	0.9922	-7.86
Subcritical(114.0cm)	Average k_{eff}	Average ρ_{eff}
G4STORK	0.8995 $\pm$ 0.0022	-79.91
MCNP	0.9625 $\pm$ 0.0007	-38.96
Experiment	0.9691	-31.88

TABLE 6.2: Effective multiplication and reactivity of the G4-STORK, MCNP codes and the Experimental measurements.

6.1.1 Critical Height

The moderator height at 132.7 cm represents the critical state of the ZED-2 reactor for this specific experiment. All the simulated k_{eff} and ρ_{eff} values from the G4-STORK and MCNP, as well as the experimental measurements, are shown in Table 6.3. As can be seen in Table 6.3, the calculated results from G4-STORK show a noticeable discrepancy between both the experimental and MCNP results. The critical height is about $\sim$-26 mk off from the experimental measurements. However, the trend of k_{eff} as function of moderator height seems to be the same as the experimental measurements and the MCNP results, as shown in Figure 6.8. Figure 6.8 demonstrates the comparison of the k_{eff} values as a function of moderator height for G4STORK, MCNP, and experimental measurements. Thus, the critical height was calculated by extrapolating of fitting a linear equation to the data points for both G4STORK and MCNP. Using extrapolation, the critical height for the G4-STORK calculations appeared to be at 141.26 cm, which

is about 8.56 cm higher than the experimental critical height ($\sim$ 6.25% difference). The G4STORK simulation for this height (141.27 cm) computed an average k_{eff} value of 1.0003.

The same method of extrapolation was used to find the critical height for the MCNP data points. The critical height for the MCNP appeared to be at 1134.26 cm which is about 1.55 cm higher than the experimental critical height ($\sim$ 1.16 % difference).

6.1.2 Subcritical Height

It can be noticed in Table 6.3 and Figure 6.8, that the discrepancy between experimental measurements and G4-STORK reactivity results gets larger as the moderator height decreases. The first subcritical height at 127.7 cm showed a difference of about $\sim$-36

Moderator Height	G4STORK	MCNP
Critical (132.7 cm)	2.53 %	0.24 %
Subcritical (127.7 cm)	3.42 %	0.29 %
Subcritical (114.0 cm)	7.19%	0.68 %

TABLE 6.3: Percentage differences between the experimental measurements and G4STORK and MCNP.

mk from the experimental measurements. And the second subcritical height at 114.0 cm showed a difference of about $\sim$-80 mk from the experimental measurements.

The discrepancies in the results were expected given the differences in the model definitions (geometry and the materials), the difference between the cross section data libraries, and the differences in the calculation methods. There are some means to improve the k_{eff} calculations. Such methods are to be developed further in the near future. However, a relatively simple way to improve the k_{eff} results was to construct a quarter core of ZED-2 reactor to obtain a better statistic with saving computational time. The next section presents the data for the quarter core.

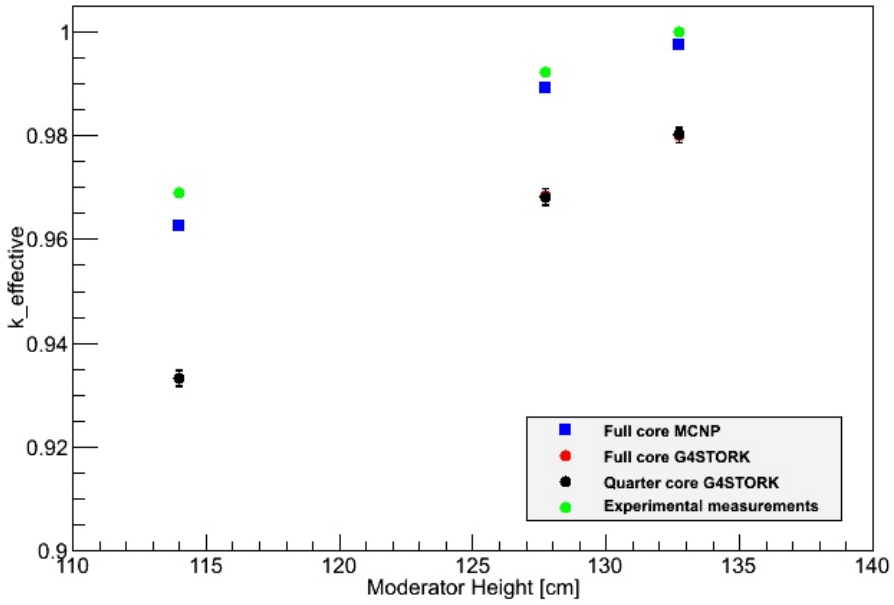


FIGURE 6.8: k_{eff} versus the moderator height for G4-STORK, MCNP, and experimental measurements.

6.2 ZED-2 Quarter Core

Since the ZED-2 core lattice that was used for this experiment has a symmetric lattice configuration, a quarter core of ZED-2 reactor was modeled in the G4STORK. Using the quarter core allowed running more primary neutrons and saving computational time to get better statistics for the calculations.

Using the quarter core required the use of boundary conditions. However, the only available boundary condition in G4-STORK at the time was the periodic boundary conditions. The periodic boundary conditions seemed not to serve the purpose for these calculations. Thus, the G4-STORK boundary condition class was modified to offer versatile boundary conditions. With the new G4-STORK boundary condition class, it was possible to make only the boundaries that are the continuation of the core as reflective boundaries while the rest of the boundaries stay as the regular boundary or zero boundary. This section presents the results for the three moderator heights as were calculated in full core. These simulations were obtained using 9×10^5 primary neutrons per run for

a period of 1ms, lasting for 200 runs. Figure 6.9 presents the k_{eff} values in time on left and k_{run} values in time on right. The first 50 runs were discarded for the convergence purposes. The survivor neutrons' information at the end of each run was saved for the

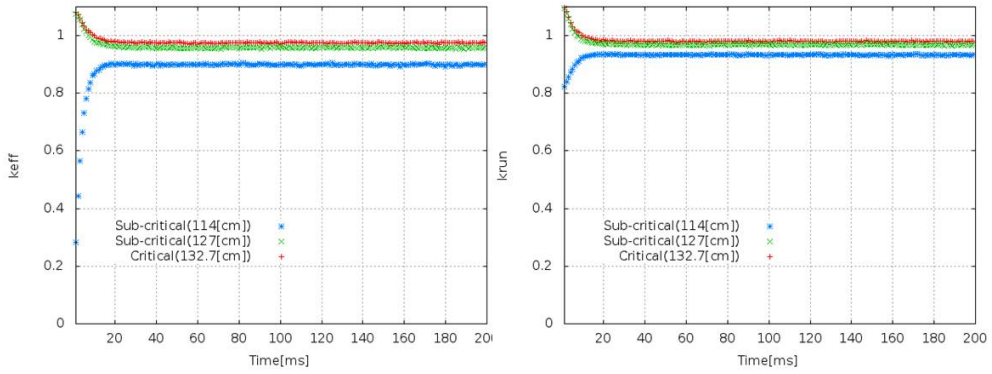


FIGURE 6.9: The effective (on left) and the run (on right) multiplication factors in time.

critical height simulation to study and show the time evolution of the neutron distribution in time. Figures 6.10 and 6.11 show the neutron spatial and energy spectrum at 1 ms, 10 ms, 15 ms, 30 ms, 50 ms, and 200 ms, respectively.

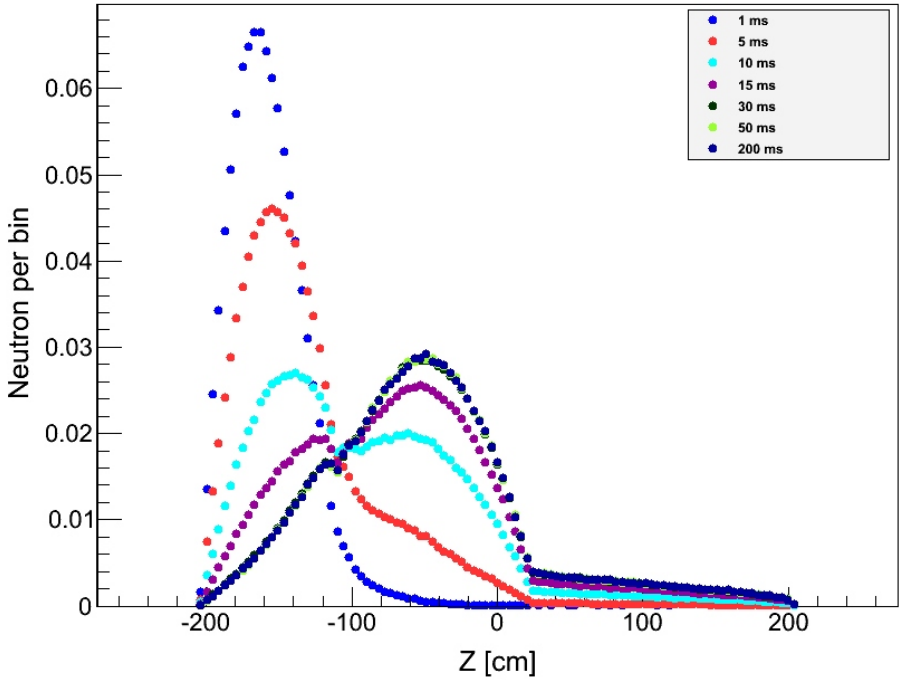


FIGURE 6.10: Neutron spatial distribution in time.

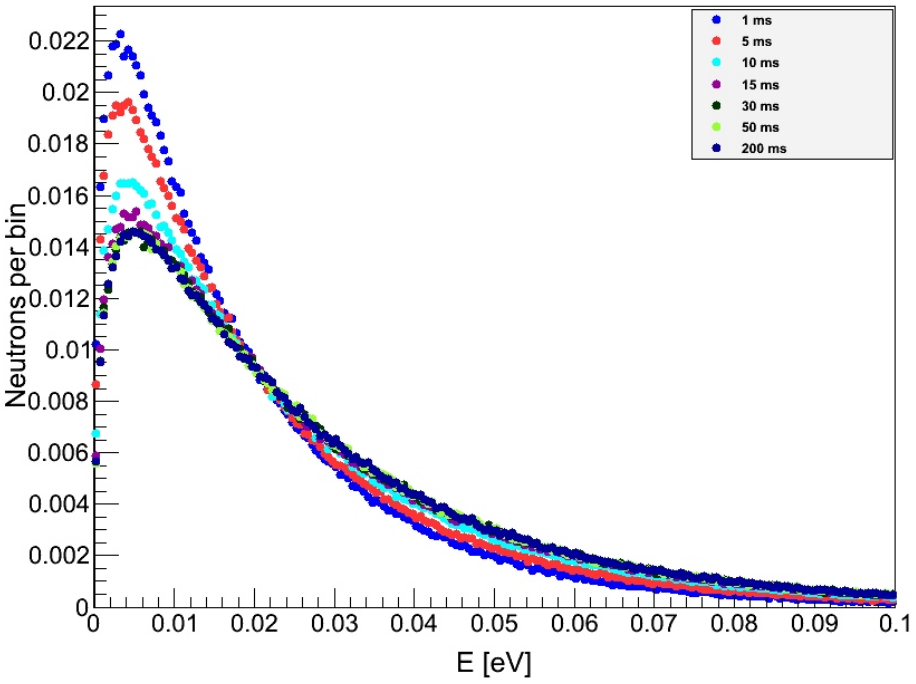
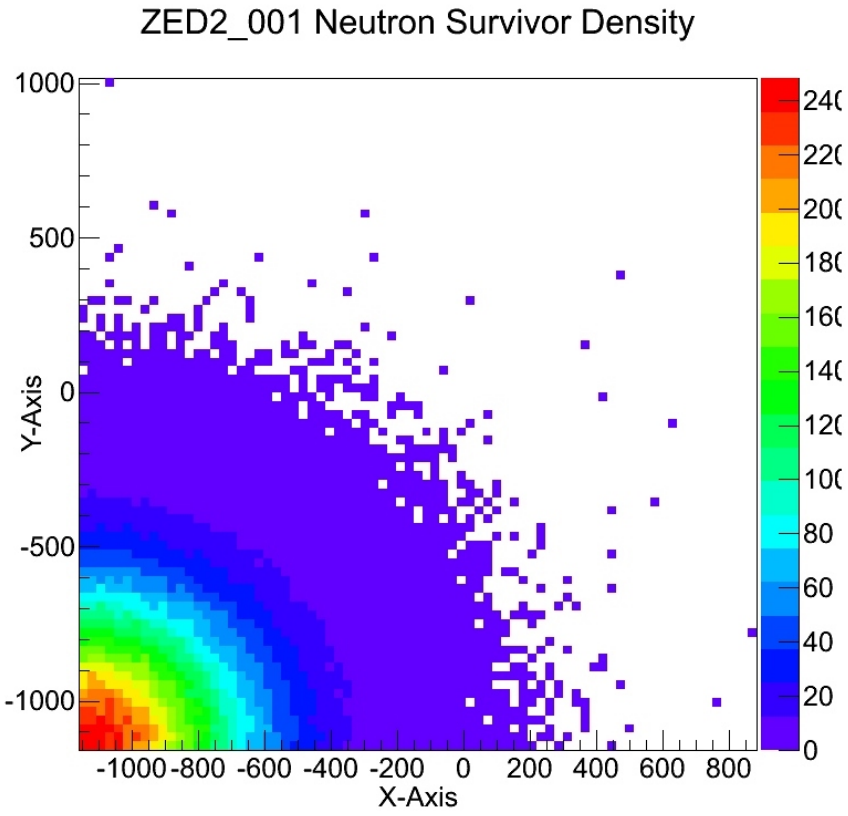


FIGURE 6.11: Energy spectrum in time.

It can be seen in these figures, that the neutron distribution and the energy spectra evolve in time and stabilize after the convergence at ∼50ms. The neutron distribution in time is shown in the animation 6.2 for ∼0.1s.



The XY-plane view (on left) and the flux map (on right) of the quarter core are shown in Figure 6.12. As expected, most of the neutrons are accumulated at the enter of the core with the right boundary conditions. The fission site locations were also collected by end of the simulation and were plotted as shown in Figure 6.13.

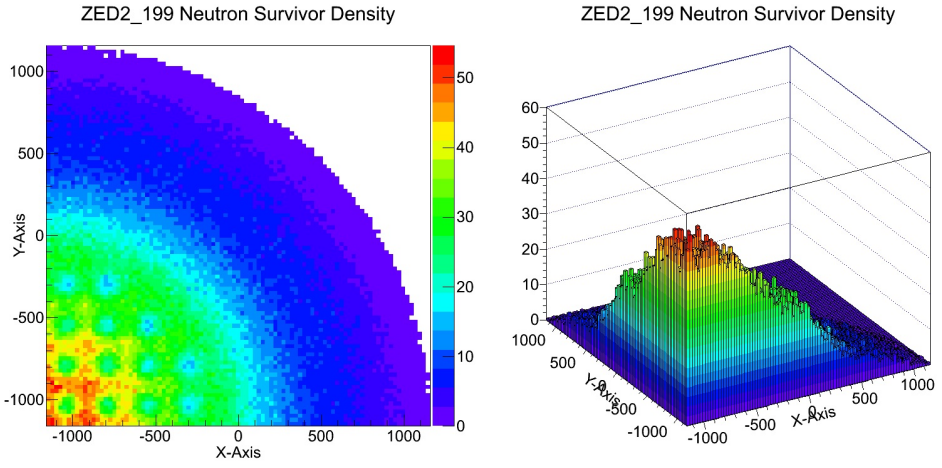


FIGURE 6.12: Flux distribution in the right.

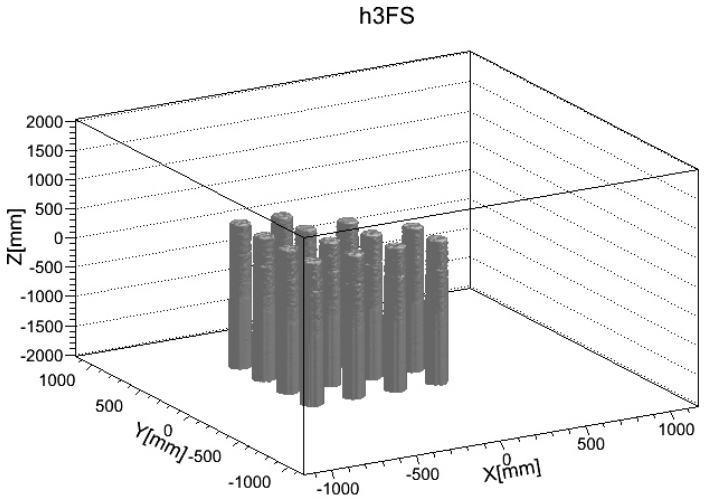


FIGURE 6.13: Fission site locations in xy plane view on left and flux distribution on the right.

Table 6.4 lists the results of k_{eff} and k_{run} for the quarter core and the full core. The results from the quarter core and full core are consistent and the differences are in unit of tenth of mk. Thus, it can be concluded that using the quarter core did save computational time but that the outcome was unaffected from the full core to quarter core.

	Full Core		Quarter Core	
Height	Average k_{eff}	Average k_{run}	Average k_{eff}	Average k_{run}
Critical (132.7 cm)	0.9747 ± 0.0018	0.9799 ± 0.0014	0.9750 ± 0.0019	0.9803 ± 0.0015
Subcritical (127.7 cm)	0.9582 ± 0.0019	0.9683 ± 0.0014	0.9579 ± 0.0023	0.9681 ± 0.0013
Subcritical (114.0 cm)	0.8995± 0.0022	0.9332 ± 0.0013	0.8994± 0.0022	0.9332 ± 0.0013

TABLE 6.4: Effective multiplication and run multiplication of full core and quarter core.

The concern of the large discrepancies in the calculations of k_{eff} may be due to the Geant4 cross section libraries G4NDL. As was discussed in section 5.3.2.5, G4NDL cross section data is at the 0 K. Therefore, extra calculations are required to Doppler broaden the cross sections for any higher temperature. It is also possible that by using the newer version of Geant4 (version 10) that many physics treatments change the results for k_{eff}. These uncertainties need to be resolved before making further conclusions on the performance of the G4-STORK code.

Chapter 7

Summary and Conclusions

This project was intended to model the subcritical reactivity measurements in ZED-2 using G4STORK reactor physics computer code to serve fulfilling a few purposes:

1. To test the performance of G4-STORK computer code for simulations of kinetic reactor physics;

2. To further improve the G4-STORK computer code for better reactor kinetics calculations according to the comparison between the simulated results by G4-STORK, MCNP, and the experimental measurements.

Three criticality calculations were performed for critical (132.7 cm) and two sub-critical (127.7 cm and 114.0 cm) moderator heights in each ZED-2 full and quarter core. The criticality calculations in quarter core were done to save the computational time to run more primary neutrons per run. However, the calculated k_{eff} values were consistent with the full core.

The calculated k_{eff} values by G4STORK showed large discrepancies compared to the results of MCNP and to the experimental measurements, as was discussed in preceding chapter 6. These discrepancies are increasing (from $\sim$26 mk to $\sim$48 mk) as the reactor become more subcritical.

The obtained k_{eff} at the critical height still shows a large discrepancy and should be investigated more carefully. This difference can be assigned to the different methodologies used to compute the cross section libraries in Geant4 (G4NDL):

1. The on-the-fly technique that is used to Doppler broaden the cross section data to the temperature of the local medium at the energy of the scattering neutron as was discussed in section 5.3.2.5.

2. Also the different methodologies used to compute the final states in Geant4 (G4NDL)

As mentioned earlier, on-the-fly Doppler broadening technique can have up to 3% of error which is a large error in terms of accuracy. Therefore, it is preferred to use the cross section data library that is evaluated near the simulated material temperature before start of the simulation in G4-STORK. In this way, the difference between the initial and the final temperatures is not as large. Thus, this way makes the interpolation error much less that have a better affect on the accuracy of the data and faster computational time.

At this end, the following remarks can be concluded from this work here:

- The subcritical reactivity measurements in ZED-2 were successfully modeled using G4-STORK as a refinement from the preliminary conceptual modeling which was used to model the real reactor physics experiment.

- The k_{eff} values obtained by G4STORK show large discrepancies for all moderator heights.

It was mentioned before that the G4-STORK is still at an early stage of its development and that this project was the first real life reactor calculations modeled using this computer code. Given the flexibility and the capability of G4-STORK, there is much opportunity to improve and extend this code. Thus, the following task should be completed to improve the results of this particular project:

1. Using the converted MCNP cross section data near 300 K to G4NDL library.

Appendix A

Full Core ZED2 Construction

```
/*
Source code for the ZED2 geometry and materials
*/
#include "ZED2Constructor.hh"
// Constructor
ZED2Constructor::ZED2Constructor()
:StorkVWorldConstructor(), tankLogical1(0),
logicRodA1(0), logicRodB1(0),logicCoolant1(0),
logicPressure1(0), logicGasAnn1(0)
{

}
// Destructor
ZED2Constructor:: ZED2Constructor()
{
// Delete visualization attributes
delete vesselVisAtt;
delete tank1VisATT;
```

```
delete ModVisAtt;
delete fuelA1VisATT;
delete fuelB1VisATT;
delete sheathA1VisATT;
delete sheathB1VisATT;
delete Air1VisAtt;
delete Coolant1VisAtt;
delete Pressure1VisAtt;
delete GasAnn1VisAtt;
delete Calandria1VisAtt;
delete EndPlate2VisATT;
delete airTubeLogical;
delete DumplineAlVisAtt;
delete DumplineHWVisAtt;
}
// ConstructNewWorld()
G4VPhysicalVolume* ZED2Constructor::ConstructNewWorld(const StorkParseInput*
infile) {

// Call base class ConstructNewWorld() to complete construction
return StorkVWorldConstructor::ConstructNewWorld(infile);
}



// ConstructWorld
// Construct the geometry and materials of the reactor given the inputs.
G4VPhysicalVolume* ZED2Constructor::ConstructWorld()
{
```

```
// Set local variables and enclosed world dimensions
reactorDim = G4ThreeVector(0.*cm, 231.806*cm ,405.4*cm/2.);
G4double buffer = 1.0*cm;
encWorldDim = G4ThreeVector(2*reactorDim[1]+buffer, 2*reactorDim[1]+buffer, 2*re-
actorDim[2]+buffer);
G4SolidStore* theSolids = G4SolidStore::GetInstance();

//Defining the graphite wall and bottom
G4double Graphitewall[3] =171.806*cm, 231.806*cm, 315.4*cm/2.;
G4double Graphitebott[3] = 0., 231.806*cm,90.0*cm/2.;

// Create Dimensions of Calandria Tank
G4double CalandriaDim1[3] =  0.*cm, 168.635*cm, 315.4*cm/2.-2.69*cm/2.;
G4double BotReacTankDim[3] = 0.*cm, 168.635*cm, 2.69*cm/2.;

// Defining the dimensions of the moderator
G4double ModHeight = 132.707*cm;
G4double distbtwflrtofuel = 10.1124*cm;
G4double RodHeight = 2.0*CalandriaDim1[2]-distbtwflrtofuel;
G4double MTankDim[3] = 0.*cm, 168.0*cm, ModHeight;
G4double TubeAirFuel[3] = 0.0*cm, 168.0*cm, (2*CalandriaDim1[2]-ModHeight)/2.;

// Create Dimensions of Fuel Assembly
G4double CalendriaT1Dim[3] = 0.0*cm, 12.74*cm, RodHeight;
G4double GasAnn1Dim[3] = 0.0*cm, 12.46*cm, RodHeight;
G4double PressureT1Dim[3] = 0.0*cm, 10.78*cm, RodHeight;
```

```
G4double Coolant1Dim[3] = 0.0*cm, 10.19*cm, (5.*(49.51*cm));
G4double Air1Dim[3] = 0.0*cm, 10.19*cm, RodHeight-(5.*(49.51*cm));
G4double EndPlate2[3] =  0.0*cm, 4.585*cm, 0.16*cm/2.0;
G4double FuelRodADim1[3] = 0.0*cm, 1.264*cm,48.25*cm/2.;
G4double FuelRodBDim1[3] = 0.0*cm, 1.070*cm,48.0*cm/2.;
G4double SheathADim1[3] = 0.0*cm, 1.350*cm, 49.19*cm/2.;
G4double SheathBDim1[3] = 0.0*cm, 1.150*cm, 49.19*cm/2.;


// Create the ring for fuel pins placement
G4int rings = 4;
G4double ringRad[3] = 1.734*cm,3.075*cm,4.384*cm;
G4double secondRingOffset = 0.261799*radian;


// Calculating the fuel cuts in the moderator and air
G4double topCalandriatoModH = 2.*CalandriaDim1[2]-ModHeight;
G4double AirinCT = RodHeight-Coolant1Dim[2];
G4double topFueltoModH = topCalandriatoModH-AirinCT;
G4double FuelinModH = Coolant1Dim[2]-(topFueltoModH);
G4int ModFuelIntersectPin = floor((((topFueltoModH)/10.)/49.51*cm)/10.);
G4int NumOfFuelBunInMod = floor((((FuelinModH)/10.)/49.51*cm)/10.);
G4double FullFuelBunInAir = ModFuelIntersectPin*49.51*cm;
G4double ModFuelIntersectPos = (topFueltoModH-FullFuelBunInAir-0.16*cm);
G4double CutFuelBunInMod = 49.19*cm-ModFuelIntersectPos;


// Create Dimensions of dump lines in graphite
G4double DumpLineAlDim[3] = 0.0*cm, 22.86*cm, 90.*cm/2.;
```

```
G4double DumplineHWDim[3] = 0.0*cm, 22.066*cm, 90.*cm/2.;


// Create Dimensions of dump lines in Al calandria
G4double DumpLineAlDimC[3] = 0.0*cm, 22.86*cm, 2.69*cm/2.;
G4double DumplineHWDimC[3] = 0.0*cm, 22.066*cm, 2.69*cm/2.;


// Positions of the fuel bundles
G4double Pich[2] = 24.5*cm, 24.5*cm;


G4double XPos[] = {
Pich[0]/2,Pich[0]/2,Pich[0]/2,Pich[0]/2,
3*Pich[0]/2, 3*Pich[0]/2, 3*Pich[0]/2, 3*Pich[0]/2,
5*Pich[0]/2,5*Pich[0]/2,5*Pich[0]/2,
7*Pich[0]/2,7*Pich[0]/2,
-Pich[0]/2,-Pich[0]/2,-Pich[0]/2,-Pich[0]/2,
-3*Pich[0]/2, -3*Pich[0]/2, -3*Pich[0]/2, -3*Pich[0]/2,
-5*Pich[0]/2,-5*Pich[0]/2,-5*Pich[0]/2,
-7*Pich[0]/2,-7*Pich[0]/2,
-Pich[0]/2,-Pich[0]/2,-Pich[0]/2,-Pich[0]/2,
-3*Pich[0]/2, -3*Pich[0]/2, -3*Pich[0]/2, -3*Pich[0]/2,
-5*Pich[0]/2,-5*Pich[0]/2,-5*Pich[0]/2,
-7*Pich[0]/2,-7*Pich[0]/2,
Pich[0]/2,Pich[0]/2,Pich[0]/2,Pich[0]/2,
3*Pich[0]/2, 3*Pich[0]/2, 3*Pich[0]/2, 3*Pich[0]/2,
5*Pich[0]/2,5*Pich[0]/2,5*Pich[0]/2,
7*Pich[0]/2,7*Pich[0]/2};
```

```
G4double YPos[] = {
Pich[1]/2, 3.*Pich[1]/2, 5.*Pich[1]/2, 7.*Pich[1]/2,
Pich[1]/2, 3.*Pich[1]/2, 5.*Pich[1]/2, 7.*Pich[1]/2,
Pich[1]/2, 3.*Pich[1]/2, 5.*Pich[1]/2,
Pich[1]/2, 3.*Pich[1]/2,
Pich[1]/2, 3.*Pich[1]/2, 5.*Pich[1]/2, 7.*Pich[1]/2,
Pich[1]/2, 3.*Pich[1]/2, 5.*Pich[1]/2, 7.*Pich[1]/2,
Pich[1]/2, 3.*Pich[1]/2, 5.*Pich[1]/2,
Pich[1]/2, 3.*Pich[1]/2,
-Pich[1]/2, -3.*Pich[1]/2, -5.*Pich[1]/2, -7.*Pich[1]/2,
-Pich[1]/2, -3.*Pich[1]/2, -5.*Pich[1]/2, -7.*Pich[1]/2,
-Pich[1]/2, -3.*Pich[1]/2, -5.*Pich[1]/2,
-Pich[1]/2, -3.*Pich[1]/2,
-Pich[1]/2, -3.*Pich[1]/2, -5.*Pich[1]/2, -7.*Pich[1]/2,
-Pich[1]/2, -3.*Pich[1]/2, -5.*Pich[1]/2, -7.*Pich[1]/2,
-Pich[1]/2, -3.*Pich[1]/2, -5.*Pich[1]/2,
-Pich[1]/2, -3.*Pich[1]/2};


// Set up the materials (if necessary)
if(matChanged)
{
// Delete any existing materials
DestroyMaterials();
// Create the materials
ConstructMaterials();
}
// Clean up volumes
G4GeometryManager::GetInstance()->OpenGeometry();
```

```
G4PhysicalVolumeStore::GetInstance()->Clean();
G4LogicalVolumeStore::GetInstance()->Clean();

// Set up the solids if necessary
if(geomChanged)
{
// Clean up solids
G4SolidStore::GetInstance()->Clean();
// Clean up solids
G4SolidStore::GetInstance()->Clean();

// Create world solid
new G4Box("ZED2World", encWorldDim[0]/2, encWorldDim[1]/2, encWorldDim[2]/2);
// Create the air above the moderator
new G4Tubs("AirTube", TubeAirFuel[0], TubeAirFuel[1], TubeAirFuel[2], 0., 2.0*CLHEP::pi);

// Create Graphite Reflector solid
new G4Tubs("graphitewall", reactorDim[0], reactorDim[1], Graphitewall[2], 0., 2.0*CLHEP::pi);
new G4Tubs("graphitebott", reactorDim[0], reactorDim[1], Graphitebott[2], 0., 2.0*CLHEP::pi);
new G4UnionSolid("graphitewall+graphitebott", theSolids->GetSolid("graphitewall"),
theSolids->GetSolid("graphitebott"), 0, G4ThreeVector(0.,0.,-Graphitewall[2]-Graphitebott[2]));

// Create Sheilding walls
//new G4Tubs("sheildingwall", Shieldingwall[0], Shieldingwall[1], Shieldingwall[2],
0., 2.0*CLHEP::pi);
// Create the Calandria solids 1
new G4Tubs("calandriashell", CalandriaDim1[0], CalandriaDim1[1], CalandriaDim1[2],
```

```
0., 2.0*CLHEP::pi);
new G4Tubs("calandriabott", BotReacTankDim[0], BotReacTankDim[1], BotReacTankDim[2],
0., 2.0*CLHEP::pi);
new G4UnionSolid("calandriashell+calandriabott", theSolids->GetSolid("calandriashell"),
theSolids->GetSolid("calandriabott"), 0, G4ThreeVector(0,0,(-CalandriaDim1[2]-BotReacTankDim[2]

// Create Moderator solid
new G4Tubs("ModSphere", MTankDim[0], MTankDim[1], MTankDim[2]/2., 0., 2.0*CLHEP::pi);

// Create the air above the coolant tube solid
new G4Tubs("AirTube1", Air1Dim[0]/2, Air1Dim[1]/2, Air1Dim[2]/2., 0., 2.0*CLHEP::pi);
// Create the Calandria tube
//new G4Tubs("CalandriaTubedwn", CalendriaT1Dim[0]/2, CalendriaT1Dim[1]/2, Cal-
endriaT1Dim[2], 0., 2.0*CLHEP::pi);
new G4Tubs("CalandriaTubedwnCut1", CalendriaT1Dim[0]/2, CalendriaT1Dim[1]/2,
(CalendriaT1Dim[2]-topCalandriatoModH)/2., 0., 2.0*CLHEP::pi);
new G4Tubs("CalandriaTubedwnCut2", CalendriaT1Dim[0]/2, CalendriaT1Dim[1]/2,
topCalandriatoModH/2., 0., 2.0*CLHEP::pi);

// Create the GasAnn tube solid
//new G4Tubs("GasAnnTube1", GasAnn1Dim[0]/2, GasAnn1Dim[1]/2, GasAnn1Dim[2],
0., 2.0*CLHEP::pi);
new G4Tubs("GasAnnTube1Cut1", GasAnn1Dim[0]/2, GasAnn1Dim[1]/2, (GasAnn1Dim[2]-
topCalandriatoModH)/2., 0., 2.0*CLHEP::pi);
new G4Tubs("GasAnnTube1Cut2", GasAnn1Dim[0]/2, GasAnn1Dim[1]/2, topCalan-
driatoModH/2., 0., 2.0*CLHEP::pi);
```

```
// Create the pressure tube solid
//new G4Tubs("PressureTubedwn", PressureT1Dim[0]/2, PressureT1Dim[1]/2, PressureT1Dim[2],
0., 2.0*CLHEP::pi);
new G4Tubs("PressureTubedwnCut1", PressureT1Dim[0]/2, PressureT1Dim[1]/2, (GasAnn1Dim[2]-
topCalandriatoModH)/2, 0., 2.0*CLHEP::pi);
new G4Tubs("PressureTubedwnCut2", PressureT1Dim[0]/2, PressureT1Dim[1]/2, top-
CalandriatoModH/2., 0., 2.0*CLHEP::pi);


// Create the coolant tube solid
//new G4Tubs("CoolantTube1", Coolant1Dim[0]/2, Coolant1Dim[1]/2, Coolant1Dim[2],
0., 2.0*CLHEP::pi);
new G4Tubs("CoolantTube1Cut1", Coolant1Dim[0]/2, Coolant1Dim[1]/2, FuelinModH/2./*-
topFueltoModH+(CalendriaT1Dim[2]-Coolant1Dim[2]) */, 0., 2.0*CLHEP::pi);
new G4Tubs("CoolantTube1Cut2", Coolant1Dim[0]/2, Coolant1Dim[1]/2, topFueltoModH/2./*(Calen
Coolant1Dim[2])/2+Coolant1Dim[2]-topFueltoModH*/, 0., 2.0*CLHEP::pi);


// Create outer fuel bunndles solid
new G4Tubs("FuelTubeB1", FuelRodBDim1[0]/2, FuelRodBDim1[1]/2, FuelRodBDim1[2],
0., 2.0*CLHEP::pi);
new G4Tubs("FuelTubeB1Cut1", FuelRodBDim1[0]/2, FuelRodBDim1[1]/2, CutFuel-
BunInMod/2., 0., 2.0*CLHEP::pi);
new G4Tubs("FuelTubeB1Cut2", FuelRodBDim1[0]/2, FuelRodBDim1[1]/2, ModFu-
elIntersectPos/2., 0., 2.0*CLHEP::pi);


// Create inner fuel bunndles solid
new G4Tubs("FuelTubeA1", FuelRodADim1[0]/2, FuelRodADim1[1]/2, FuelRodADim1[2],
0., 2.0*CLHEP::pi);
```

```
new G4Tubs("FuelTubeA1Cut1", FuelRodADim1[0]/2, FuelRodADim1[1]/2, CutFuelBunInMod/2., 0., 2.0*CLHEP::pi);
new G4Tubs("FuelTubeA1Cut2", FuelRodADim1[0]/2, FuelRodADim1[1]/2, ModFuelIntersectPos/2., 0., 2.0*CLHEP::pi);


// Create outer Zr-4 sheath solid
new G4Tubs("SheathB1", SheathBDim1[0]/2, SheathBDim1[1]/2, SheathBDim1[2], 0., 2.0*CLHEP::pi);
new G4Tubs("SheathB1Cut1", SheathBDim1[0]/2, SheathBDim1[1]/2, CutFuelBunInMod/2., 0., 2.0*CLHEP::pi);
new G4Tubs("SheathB1Cut2", SheathBDim1[0]/2, SheathBDim1[1]/2, ModFuelIntersectPos/2., 0., 2.0*CLHEP::pi);


// Create inner Zr-4 sheath solid
new G4Tubs("SheathA1", SheathADim1[0]/2, SheathADim1[1]/2, SheathADim1[2], 0., 2.0*CLHEP::pi);
new G4Tubs("SheathA1Cut1", SheathADim1[0]/2, SheathADim1[1]/2, CutFuelBunInMod/2., 0., 2.0*CLHEP::pi);
new G4Tubs("SheathA1Cut2", SheathADim1[0]/2, SheathADim1[1]/2, ModFuelIntersectPos/2., 0., 2.0*CLHEP::pi);


// Create the end plate
new G4Tubs("EndPlate2", EndPlate2[0], EndPlate2[1], EndPlate2[2], 0., 2.0*CLHEP::pi);
new G4Tubs("EndPlate1", EndPlate2[0], EndPlate2[1], EndPlate2[2], 0., 2.0*CLHEP::pi);
// Create Dump Line solid in graphite
new G4Tubs("DumpLineAl", DumpLineAlDim[0]/2, DumpLineAlDim[1]/2, DumpLineAlDim[2], 0., 2.0*CLHEP::pi);
```

```
new G4Tubs("DumpLineHW", DumplineHWDim[0]/2, DumplineHWDim[1]/2, Dumpline-
HWDim[2], 0., 2.0*CLHEP::pi);
// Create Dump Line solid in Al Calandria
new G4Tubs("DumpLineAlC", DumpLineAlDimC[0]/2, DumpLineAlDimC[1]/2, DumpLin-
eAlDimC[2], 0., 2.0*CLHEP::pi);
new G4Tubs("DumpLineHWC", DumplineHWDimC[0]/2, DumplineHWDimC[1]/2,
DumplineHWDimC[2], 0., 2.0*CLHEP::pi);


geomChanged = false;
}


// Create world volume
// Create world volume
worldLogical = new G4LogicalVolume(theSolids->GetSolid("ZED2World"),matMap["World"],
"worldLogical",0,0,0); worldPhysical = new G4PVPlacement(0, G4ThreeVector(0.,0.,0.),
worldLogical,"worldPhysical",0,0,0);


// Create Reflector volume
vesselLogical = new G4LogicalVolume(theSolids->GetSolid("graphitewall+graphitebott"),
matMap["Graphite"], "VesselLogical", 0, 0, 0);
new G4PVPlacement(0,G4ThreeVector(0,0,Graphitebott[2]), vesselLogical, "VesselPhys-
ical", worldLogical, 0, 0);


// Create Calandrai volume in mother air volume tankLogical1 = new G4LogicalVolume(theSolids-
>GetSolid("calandriashell+calandriabott"), matMap["Al57S"], "VesselLogical1", 0, 0,
0);
new G4PVPlacement(0, G4ThreeVector(0.,0.,BotReacTankDim[2]), tankLogical1,"CalandriaPhysical
```

```
vesselLogical,0,0);


// Create Moderator volume ModLogical = new G4LogicalVolume(theSolids->GetSolid("ModSphere")
"ModLogical",0,0,0);
new G4PVPlacement(0, G4ThreeVector(0.,0.,MTankDim[2]/2.-CalandriaDim1[2]), Mod-
Logical, "ModPhysical", tankLogical1, false, 0);


//Create Air above the moderator airTubeLogical = new G4LogicalVolume(theSolids-
>GetSolid("AirTube"),matMap["Air"], "airTubeLogical",0,0,0); new G4PVPlacement(0,
G4ThreeVector(0.,0., TubeAirFuel[2]+MTankDim[2]-CalandriaDim1[2]), airTubeLog-
ical, "airTubePhysical", tankLogical1,0,0);
std::stringstream volName;


logicCalandria1 = new G4LogicalVolume(theSolids->GetSolid("CalandriaTubedwnCut2"),
matMap["AlCalT"], "CalandriaTube1LogicalCut2", 0, 0, 0);
for (G4int i=0; i<52; i++)
{
// Create calandria tubes in mother air volume
volName.str("");
volName.clear();
volName « i;
new G4PVPlacement (0, G4ThreeVector(XPos[i],YPos[i],0.), logicCalandria1, "Calan-
driaTube1PhysicalCut2"+volName.str(), airTubeLogical, false, 0);
}
// Create gas annulus tubes in mother air volume
logicGasAnn1 = new G4LogicalVolume(theSolids->GetSolid("GasAnnTube1Cut2"), matMap["Air"],
"GasAnnTube1Logical", 0, 0, 0);
```

```
new G4PVPlacement (0, G4ThreeVector(0,0,0), logicGasAnn1, "GasAnnTube1PhysicalCut2",
logicCalandria1, false, 0);


// Create pressure tubes in mother air volume
logicPressure1 = new G4LogicalVolume(theSolids->GetSolid("PressureTubedwnCut2"),
matMap["AlPresT"], "PressureTube1Logical", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,0), logicPressure1, "PressureTube1PhysicalCut2",
logicGasAnn1, false, 0);


// Create lower coolant in mother air volume
logicCoolant1 = new G4LogicalVolume(theSolids->GetSolid("CoolantTube1Cut2"), matMap["Air"],
"Coolant1Logical", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,-topCalandriatoModH/2.+topFueltoModH/2.),
logicCoolant1, "Coolant1PhysicalCut2", logicPressure1, false, 0);


// Create air
logicAir1 = new G4LogicalVolume(theSolids->GetSolid("AirTube1"), matMap["Air"],
"Air1Logical", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,-topCalandriatoModH/2.+topFueltoModH+Air1Dim[2]/2
logicAir1, "Air1Physical", logicPressure1, false, 0);


// Create inner/outer FULL fuel bunndles and sheath in air volume
logicRodA1 = new G4LogicalVolume(theSolids->GetSolid("FuelTubeA1"), matMap["LEUMat"],
"FuelRodA1Logical", 0, 0, 0);
logicRodB1 = new G4LogicalVolume(theSolids->GetSolid("FuelTubeB1"), matMap["LEUMat"],
"FuelRodB1Logical", 0, 0, 0);
logicSheathA1 = new G4LogicalVolume(theSolids->GetSolid("SheathA1"), matMap["Zr4"],
```

```
"SheathA1Logical", 0, 0, 0);
logicSheathB1 = new G4LogicalVolume(theSolids->GetSolid("SheathB1"), matMap["Zr4"],
"SheathB1Logical", 0, 0, 0);
logicEndPlate1 = new G4LogicalVolume(theSolids->GetSolid("EndPlate1"), matMap["Zr4"],
"EndPlate1", 0, 0, 0);
logicEndPlate2 = new G4LogicalVolume(theSolids->GetSolid("EndPlate2"), matMap["Zr4"],
"EndPlate2", 0, 0, 0);


for (G4int l=0; l<ModFuelIntersectPin; l++)
{
// Rotation and translation of the rod and sheathe


// place the center pin in air
new G4PVPlacement(0, G4ThreeVector(0,0, (topFueltoModH/2.-(l+1)*(SheathADim1[2]+2.*EndPlat
l*(2.*EndPlate2[2]+SheathADim1[2])) ), logicSheathA1,"sheathePhysical " + volName.str(),
logicCoolant1,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodA1,"fuelPhysicalA", logicSheathA1,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodB1,"fuelPhysicalB", logicSheathB1,0,0);


for( G4int j = 1; j < rings; j++ )
{
for( G4int k = 0; k < j*6; k++ )
{
// Reset string stream
volName.str("");
```

```
volName « j « "-" « k;


if(j == 2)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffse
ringRad[j-1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffset),(topFueltoModH/2
(l+1)*(SheathADim1[2]+2.*EndPlate2[2])-l*(2.*EndPlate2[2]+SheathADim1[2])));
new G4PVPlacement(0, Tm, logicSheathB1,"sheathePhysical " +volName.str(),logicCoolant1,0,0);


}
else if (j == 1)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),(topFueltoModH/2.-(l+1)*(SheathADim1[2]+2.*
l*(2.*EndPlate2[2]+SheathADim1[2])));
new G4PVPlacement(0, Tm, logicSheathA1,"sheathePhysical " +volName.str(),logicCoolant1,0,0);
}
else
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),(topFueltoModH/2.-(l+1)*(SheathADim1[2]+2.*
l*(2.*EndPlate2[2]+SheathADim1[2])));
new G4PVPlacement(0, Tm, logicSheathB1,"sheathePhysical " +volName.str(),logicCoolant1,0,0);
}
}
}
```

```
// Make the end plates 1
G4ThreeVector EP1(0,0,(topFueltoModH/2.-EndPlate2[2])-l*(49.51*cm)); new G4PVPlacement(0,
EP1, logicEndPlate1,"EndPlate1Physical1",logicCoolant1,0,0);
// Make the end plates 2
G4ThreeVector EP2(0,0,(topFueltoModH/2.-EndPlate2[2])-l*(2.*EndPlate2[2])-(l+1)*(2.*SheathADi
new G4PVPlacement(0, EP2, logicEndPlate2,"EndPlate2Physical1",logicCoolant1,0,0);


}


// Create inner/outer cut fuel bunndl in air volume
logicRodA1Cut2 = new G4LogicalVolume(theSolids->GetSolid("FuelTubeA1Cut2"),
matMap["LEUMat"], "FuelRodA1LogicalCut2", 0, 0, 0);
logicRodB1Cut2 = new G4LogicalVolume(theSolids->GetSolid("FuelTubeB1Cut2"),
matMap["LEUMat"], "FuelRodB1LogicalCut2", 0, 0, 0);
logicSheathA1Cut2 = new G4LogicalVolume(theSolids->GetSolid("SheathA1Cut2"),
matMap["Zr4"], "SheathA1Logicalcut2", 0, 0, 0);
logicSheathB1Cut2 = new G4LogicalVolume(theSolids->GetSolid("SheathB1Cut2"),
matMap["Zr4"], "SheathB1LogicalCut2", 0, 0, 0);
logicEndPlate2Cut2 = new G4LogicalVolume(theSolids->GetSolid("EndPlate2"), matMap["Zr4"],
"EndPlate2Cut2", 0, 0, 0);
// Rotation and translation of the rod and sheathe


// Set name for sheathe physical volume


volName.str("");
volName « 0;
```

```
// place the center pin in air
new G4PVPlacement(0, G4ThreeVector(0,0,-topFueltoModH/2.+ModFuelIntersectPos/2.),
logicSheathA1Cut2,"sheathePhysicalCut2 " + volName.str(), logicCoolant1,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodA1Cut2,"fuelPhysicalCut2A",
logicSheathA1Cut2,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodB1Cut2,"fuelPhysicalCut2B",
logicSheathB1Cut2,0,0);


for( G4int j = 1; j < rings; j++ )
{
for( G4int k = 0; k < j*6; k++ )
{
// Reset string stream
volName.str("");


volName « j « "-" « k;


if(j == 2)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffse
ringRad[j-1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffset),-topFueltoModH/2
new G4PVPlacement(0, Tm, logicSheathB1Cut2,"sheathePhysicalCut2 " +volName.str(),logicCoolant


}
else if (j == 1)
{
```

```
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),-topFueltoModH/2.+ModFuelIntersectPos/2.);
new G4PVPlacement(0, Tm, logicSheathA1Cut2,"sheathePhysicalCut2 " +volName.str(),logicCoolant
}
else
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),-topFueltoModH/2.+ModFuelIntersectPos/2.);
new G4PVPlacement(0, Tm, logicSheathB1Cut2,"sheathePhysicalCut2 " +volName.str(),logicCoolant
}
}
}

// Make the end plates 2
G4ThreeVector EP2(0,0,-topFueltoModH/2.+ModFuelIntersectPos+0.08*cm); new G4PVPlacement(0
EP2, logicEndPlate2Cut2,"EndPlate2Physical1Cut2",logicCoolant1,0,0);

// *********Create Calandrai volume in moderator volume*************
logicCalandria1Mod = new G4LogicalVolume(theSolids->GetSolid("CalandriaTubedwnCut1"),
matMap["AlCalT"], "CalandriaTube1ModLogical", 0, 0, 0);
for (G4int i=0; i<52; i++)
{
// Create calandria tubes in moderator volume
volName.str("");
volName.clear();
volName « i;
new G4PVPlacement (0, G4ThreeVector(XPos[i],YPos[i],distbtwflrtofuel/2.), logicCa-
landria1Mod, "CalandriaTube1ModPhysicalCut1"+volName.str(), ModLogical, false,
```

```
0);
}
// Create gas annulus tubes in moderator volume
logicGasAnn1Mod = new G4LogicalVolume(theSolids->GetSolid("GasAnnTube1Cut1"),
matMap["Air"], "GasAnnTube1Logical", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,0), logicGasAnn1Mod, "GasAnnTube1PhysicalCut1",
logicCalandria1Mod, false, 0);

// Create pressure tubes in moderator volume
logicPressure1Mod = new G4LogicalVolume(theSolids->GetSolid("PressureTubedwnCut1"),
matMap["AlPresT"], "PressureTube1Logical", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,0), logicPressure1Mod, "PressureTube1PhysicalCut1",
logicGasAnn1Mod, false, 0);

// Create lower coolant in moderator volume
logicCoolant1Mod = new G4LogicalVolume(theSolids->GetSolid("CoolantTube1Cut1"),
matMap["Air"], "Coolant1Logical", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,0), logicCoolant1Mod, "Coolant1PhysicalCut1",
logicPressure1Mod, false, 0);

// Create inner/outer fuel bunndle and sheath in Moderator volume
logicRodA1Mod = new G4LogicalVolume(theSolids->GetSolid("FuelTubeA1"), matMap["LEUMat"],
"FuelRodA1LogicalMod", 0, 0, 0);
logicRodB1Mod = new G4LogicalVolume(theSolids->GetSolid("FuelTubeB1"), matMap["LEUMat"],
"FuelRodB1LogicalMod", 0, 0, 0);
logicSheathA1Mod = new G4LogicalVolume(theSolids->GetSolid("SheathA1"), matMap["Zr4"],
"SheathA1LogicalMod", 0, 0, 0);
```

```
logicSheathB1Mod = new G4LogicalVolume(theSolids->GetSolid("SheathB1"), matMap["Zr4"],
"SheathB1LogicalMod", 0, 0, 0);
logicEndPlate1Mod = new G4LogicalVolume(theSolids->GetSolid("EndPlate1"), matMap["Zr4"],
"EndPlate1Mod", 0, 0, 0);
logicEndPlate2Mod = new G4LogicalVolume(theSolids->GetSolid("EndPlate2"), matMap["Zr4"],
"EndPlate2Mod", 0, 0, 0);


for (G4int l=0; l<NumOfFuelBunInMod; l++)
{
// Rotation and translation of the rod and sheathe


// Set name for sheathe physical volume
// place the center pin
new G4PVPlacement(0, G4ThreeVector(0,0, (-FuelinModH/2.+(l+1)*(SheathADim1[2]+2.*EndPlate2
)), logicSheathA1Mod,"sheathePhysicalMod " + volName.str(), logicCoolant1Mod,0,0);
new  G4PVPlacement(0,  G4ThreeVector(0,0,0),  logicRodA1Mod,"fuelPhysicalModA
", logicSheathA1Mod,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodB1Mod,"fuelPhysicalModB ",
logicSheathB1Mod,0,0);


for( G4int j = 1; j < rings; j++ )
{
for( G4int k = 0; k < j*6; k++ )
{
// Reset string stream
volName.str("");
```

```
volName « j « "-" « k;


if(j == 2)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffse
ringRad[j-1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffset),(-FuelinModH/2.+
new G4PVPlacement(0, Tm, logicSheathB1Mod,"sheathePhysicalMod " +volName.str(),logicCoolant


}
else if (j == 1)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),(-FuelinModH/2.+(l+1)*(SheathADim1[2]+2.*E
new G4PVPlacement(0, Tm, logicSheathA1Mod,"sheathePhysicalMod " +volName.str(),logicCoolant


}
else
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),(-FuelinModH/2.+(l+1)*(SheathADim1[2]+2.*E
new G4PVPlacement(0, Tm, logicSheathB1Mod,"sheathePhysicalMod " +volName.str(),logicCoolant


}
}
}
// Make the end plates 1 G4ThreeVector EP1(0,0,(-FuelinModH/2.+EndPlate2[2])+l*(49.51*cm));
new G4PVPlacement(0, EP1, logicEndPlate1Mod,"EndPlate1Physical1Mod ",logicCoolant1Mod,0,0);
```

```
// Make the end plates 2
G4ThreeVector EP2(0,0,(-FuelinModH/2.+EndPlate2[2])+l*(2.*EndPlate2[2])+(l+1)*(2.*SheathADir
new G4PVPlacement(0, EP2, logicEndPlate2Mod,"EndPlate2Physical1Mod ",logicCoolant1Mod,0,0);
}

// Create inner/outer cut fuel bunndle in Moderator volume
logicRodA1Cut1 = new G4LogicalVolume(theSolids->GetSolid("FuelTubeA1Cut1"),
matMap["LEUMat"], "FuelRodA1LogicalCut1", 0, 0, 0);
logicRodB1Cut1 = new G4LogicalVolume(theSolids->GetSolid("FuelTubeB1Cut1"),
matMap["LEUMat"], "FuelRodB1LogicalCut1", 0, 0, 0);
logicSheathA1Cut1 = new G4LogicalVolume(theSolids->GetSolid("SheathA1Cut1"),
matMap["Zr4"], "SheathA1LogicalCut1", 0, 0, 0);
logicSheathB1Cut1 = new G4LogicalVolume(theSolids->GetSolid("SheathB1Cut1"),
matMap["Zr4"], "SheathB1LogicalCut1", 0, 0, 0);
logicEndPlate2Cut1 = new G4LogicalVolume(theSolids->GetSolid("EndPlate2"), matMap["Zr4"],
"EndPlate2Cut1", 0, 0, 0);

// place the center pin for the cut fuel bundle in the moderator
new G4PVPlacement(0, G4ThreeVector(0,0, (-FuelinModH/2.+(NumOfFuelBunInMod*49.51*cm)+2
logicSheathA1Cut1,"sheathePhysicalCut1 " + volName.str(), logicCoolant1Mod,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodA1Cut1,"fuelPhysicalCut1A
", logicSheathA1Cut1,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodB1Cut1,"fuelPhysicalCut1B
", logicSheathB1Cut1,0,0);

for( G4int j = 1; j < rings; j++ )
{
```

```
for( G4int k = 0; k < j*6; k++ )
{
// Reset string stream
volName.str("");

volName « j « "-" « k;

if(j == 2)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffse
ringRad[j-1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffset),(-FuelinModH/2.+
));
// place the fuel for the cut fuel bundle in the moderator
new G4PVPlacement(0, Tm, logicSheathB1Cut1,"sheathePhysicalCut1 " +volName.str(),logicCoolant

}
else if (j == 1)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),(-FuelinModH/2.+(NumOfFuelBunInMod*49.5
));
new G4PVPlacement(0, Tm, logicSheathA1Cut1,"sheathePhysicalCut1 " +volName.str(),logicCoolant
}
else
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),(-FuelinModH/2.+(NumOfFuelBunInMod*49.5
```

```
));
// place the fuel for the cut fuel bundle in the moderator
new G4PVPlacement(0, Tm, logicSheathB1Cut1,"sheathePhysicalCut1 " +volName.str(),logicCoolant
}
}
}

// Make the end plates 2
G4ThreeVector EP(0,0,(-FuelinModH/2.+(NumOfFuelBunInMod*49.51*cm)+0.08*cm));
new G4PVPlacement(0, EP, logicEndPlate2Cut1,"EndPlate2Physical1Cut1 ",logicCoolant1Mod,0,0);

// Create Dump Lines Al & Heavy Water in graphite
logicDumplineAl = new G4LogicalVolume(theSolids->GetSolid("DumpLineAl"), matMap["Al57S"],
"DumplineLogicalAl", 0, 0, 0);
for (G4int s=0; s<3; s++)
{
G4ThreeVector jg(129.54*cm*cos(2.0*CLHEP::pi*s*(2./3.)),129.54*cm*sin(2.0*CLHEP::pi*s*(2./3
Graphitewall[2]-2.*Graphitebott[2]+DumpLineAlDim[2]);
new G4PVPlacement (0, jg , logicDumplineAl, "DumLineAlPhysical", vesselLogical,
false, 0);
}
logicDumplineHW = new G4LogicalVolume(theSolids->GetSolid("DumpLineHW"),
matMap["Moderator"], "DumplineLogicalHW", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,0) , logicDumplineHW, "DumLineHW-
Physical", logicDumplineAl, false, 0);

// Create Dump Lines Al & Heavy Water in Al calandria
```

```
logicDumplineAlC = new G4LogicalVolume(theSolids->GetSolid("DumpLineAlC"),
matMap["Al57S"], "DumplineLogicalAlC", 0, 0, 0);
for (G4int d=0; d<3; d++)
{
//G4ThreeVector
lm(129.54*cm*cos(2.0*CLHEP::pi*d*(2./3.)),129.54*cm*sin(2.0*CLHEP::pi*d*(2./3.)),-
CalandriaDim1[2]-2.*BotReacTankDim[2]+DumpLineAlDimC);
G4ThreeVector jg(129.54*cm*cos(2.0*CLHEP::pi*d*(2./3.)),129.54*cm*sin(2.0*CLHEP::pi*d*(2./3
CalandriaDim1[2]-2.*BotReacTankDim[2]+DumpLineAlDimC[2]);
new G4PVPlacement (0, jg, logicDumplineAlC, "DumLineAlCPhysical",tankLogical1
, false, 0);
}

logicDumplineHWC = new G4LogicalVolume(theSolids->GetSolid("DumpLineHWC"),
matMap["Moderator"], "DumplineLogicalHWC", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,0) , logicDumplineHWC, "DumLineHWC-
Physical", logicDumplineAlC, false, 0);

// Set reactor as sensitive detector
worldLogical->SetSensitiveDetector( sDReactor );
airTubeLogical->SetSensitiveDetector( sDReactor );
vesselLogical->SetSensitiveDetector( sDReactor );
tankLogical1->SetSensitiveDetector( sDReactor );
ModLogical->SetSensitiveDetector( sDReactor );
logicCalandria1->SetSensitiveDetector( sDReactor );
logicGasAnn1->SetSensitiveDetector( sDReactor );
logicPressure1->SetSensitiveDetector( sDReactor );
logicCoolant1->SetSensitiveDetector( sDReactor );
```

```
logicAir1->SetSensitiveDetector( sDReactor );
logicRodA1->SetSensitiveDetector( sDReactor );
logicRodB1->SetSensitiveDetector( sDReactor );
logicSheathA1->SetSensitiveDetector( sDReactor );
logicSheathB1->SetSensitiveDetector( sDReactor );
logicEndPlate2->SetSensitiveDetector( sDReactor );
logicEndPlate1->SetSensitiveDetector( sDReactor );
logicCalandria1Mod->SetSensitiveDetector( sDReactor );
logicGasAnn1Mod->SetSensitiveDetector( sDReactor );
logicPressure1Mod->SetSensitiveDetector( sDReactor );
logicCoolant1Mod->SetSensitiveDetector( sDReactor );
logicRodA1Cut2->SetSensitiveDetector( sDReactor );
logicRodB1Cut2->SetSensitiveDetector( sDReactor );
logicSheathA1Cut2->SetSensitiveDetector( sDReactor );
logicSheathB1Cut2->SetSensitiveDetector( sDReactor );
logicEndPlate2Cut2->SetSensitiveDetector( sDReactor );
logicRodA1Mod->SetSensitiveDetector( sDReactor );
logicRodB1Mod->SetSensitiveDetector( sDReactor );
logicSheathA1Mod->SetSensitiveDetector( sDReactor );
logicSheathB1Mod->SetSensitiveDetector( sDReactor );
logicEndPlate2Mod->SetSensitiveDetector( sDReactor );
logicEndPlate1Mod->SetSensitiveDetector( sDReactor );
logicRodA1Cut1->SetSensitiveDetector( sDReactor );
logicRodB1Cut1->SetSensitiveDetector( sDReactor );
logicSheathA1Cut1->SetSensitiveDetector( sDReactor );
logicSheathB1Cut1->SetSensitiveDetector( sDReactor );
logicEndPlate2Cut1->SetSensitiveDetector( sDReactor );
logicDumplineAl->SetSensitiveDetector( sDReactor );
logicDumplineHW->SetSensitiveDetector( sDReactor );
```

```
logicDumplineAlC->SetSensitiveDetector( sDReactor );
logicDumplineHWC->SetSensitiveDetector( sDReactor );


// Set visualization attributes


worldVisAtt = new G4VisAttributes(G4Colour(0.5, 1., 0.5));
worldVisAtt->SetVisibility(true);
worldLogical->SetVisAttributes(worldVisAtt);


airTubeVisAtt = new G4VisAttributes(G4Colour(0., 1., 0.5));
airTubeVisAtt->SetVisibility(true);
airTubeLogical->SetVisAttributes(airTubeVisAtt);


vesselVisAtt= new G4VisAttributes(G4Colour(1.0,0.0,0.0));
//vesselVisAtt->SetForceSolid(true);
vesselVisAtt->SetVisibility(true);
vesselLogical->SetVisAttributes(vesselVisAtt);


tank1VisATT= new G4VisAttributes(G4Colour(1.0,1.0,0.0));
//tank1VisATT->SetForceSolid(true);
tank1VisATT->SetVisibility(true);
tankLogical1->SetVisAttributes(tank1VisATT);


ModVisAtt = new G4VisAttributes(G4Colour(0.,1.,0.));
ModVisAtt->SetVisibility(true);
//ModVisAtt->SetForceSolid(true);
```

```
ModLogical->SetVisAttributes(ModVisAtt);


Calandria1VisAtt = new G4VisAttributes(G4Colour(1., 0., 1.));
//Calandria1VisAtt->SetForceSolid(true);
Calandria1VisAtt->SetVisibility(false);
logicCalandria1->SetVisAttributes(Calandria1VisAtt);
logicCalandria1Mod->SetVisAttributes(Calandria1VisAtt);


GasAnn1VisAtt = new G4VisAttributes(G4Colour(1., 0., 0.));
// GasAnn1VisAtt->SetForceSolid(true);
GasAnn1VisAtt->SetVisibility(false);
logicGasAnn1->SetVisAttributes(GasAnn1VisAtt);
logicGasAnn1Mod->SetVisAttributes(GasAnn1VisAtt);


Pressure1VisAtt = new G4VisAttributes(G4Colour(0., 1., 0.));
// Pressure1VisAtt->SetForceSolid(true);
Pressure1VisAtt->SetVisibility(false);
logicPressure1->SetVisAttributes(Pressure1VisAtt);
logicPressure1Mod->SetVisAttributes(Pressure1VisAtt);


Coolant1VisAtt = new G4VisAttributes(G4Colour(0.53, 0.81, 0.92));
//Coolant1VisAtt->SetForceSolid(true);
Coolant1VisAtt->SetVisibility(true);
logicCoolant1->SetVisAttributes(Coolant1VisAtt);
logicCoolant1Mod->SetVisAttributes(Coolant1VisAtt);
```

```
Air1VisAtt = new G4VisAttributes(G4Colour(0., 0.5, 1.));
//Air1VisAtt->SetForceSolid(true);
Air1VisAtt->SetVisibility(true);
logicAir1->SetVisAttributes(Air1VisAtt);


fuelA1VisATT = new G4VisAttributes(G4Colour(0.0, 0.0 ,1.0));
fuelA1VisATT->SetForceSolid(true);
fuelA1VisATT->SetVisibility(true);
logicRodA1->SetVisAttributes(fuelA1VisATT);
logicRodA1Cut2->SetVisAttributes(fuelA1VisATT);
logicRodA1Mod->SetVisAttributes(fuelA1VisATT);
logicRodA1Cut1->SetVisAttributes(fuelA1VisATT);


fuelB1VisATT = new G4VisAttributes(G4Colour(0,0.5,0.92));
fuelB1VisATT->SetForceSolid(true);
fuelB1VisATT->SetVisibility(true);
logicRodB1->SetVisAttributes(fuelB1VisATT);
logicRodB1Cut2->SetVisAttributes(fuelB1VisATT);
logicRodB1Mod->SetVisAttributes(fuelB1VisATT);
logicRodB1Cut1->SetVisAttributes(fuelB1VisATT);


sheathA1VisATT = new G4VisAttributes(G4Colour(0.5, 0.0 ,1.0));
//sheathA1VisATT->SetForceSolid(true);
sheathA1VisATT->SetVisibility(true);
logicSheathA1->SetVisAttributes(sheathA1VisATT);
logicSheathA1Cut2->SetVisAttributes(sheathA1VisATT);
logicSheathA1Mod->SetVisAttributes(sheathA1VisATT);
```

```
logicSheathA1Cut1->SetVisAttributes(sheathA1VisATT);


sheathB1VisATT = new G4VisAttributes(G4Colour(1.0, 0.5 ,1.0));
//sheathB1VisATT->SetForceSolid(true);
sheathB1VisATT->SetVisibility(true);
logicSheathB1->SetVisAttributes(sheathB1VisATT);
logicSheathB1Cut2->SetVisAttributes(sheathB1VisATT);
logicSheathB1Mod->SetVisAttributes(sheathB1VisATT);
logicSheathB1Cut1->SetVisAttributes(sheathB1VisATT);


EndPlate2VisATT = new G4VisAttributes(G4Colour(0.5, 0.5, 0.5));
EndPlate2VisATT->SetForceSolid(true);
EndPlate2VisATT->SetVisibility(true);
logicEndPlate2->SetVisAttributes(EndPlate2VisATT);
logicEndPlate1->SetVisAttributes(EndPlate2VisATT);
logicEndPlate2Cut2->SetVisAttributes(EndPlate2VisATT);
logicEndPlate2Mod->SetVisAttributes(EndPlate2VisATT);
logicEndPlate1Mod->SetVisAttributes(EndPlate2VisATT);
logicEndPlate2Cut1->SetVisAttributes(EndPlate2VisATT);


DumplineAlVisAtt = new G4VisAttributes(G4Colour(1., 0.99, 0.5));
DumplineAlVisAtt->SetForceSolid(false);
DumplineAlVisAtt->SetVisibility(true);
logicDumplineAl->SetVisAttributes(DumplineAlVisAtt);
logicDumplineAlC->SetVisAttributes(DumplineAlVisAtt);
```

```
DumplineHWVisAtt = new G4VisAttributes(G4Colour(0., 1.0, 0.));
DumplineHWVisAtt->SetForceSolid(false);
DumplineHWVisAtt->SetVisibility(true);
logicDumplineHW->SetVisAttributes(DumplineHWVisAtt);
logicDumplineHWC->SetVisAttributes(DumplineHWVisAtt);


return worldPhysical;
}


// ConstructMaterials()
// Construct all the materials needed for the ZED2Constructor.
void ZED2Constructor::ConstructMaterials()
{
// Elements, isotopes and materials
G4Isotope *U234, *U235, *U238, *U236, *D2, *O16, *O17,
*Fe54, *Fe56, *Fe57, *Fe58, *Cr50, *Cr52, *Cr53, *Cr54,
*Si28, *Si29, *Si30, *Cu63, *Cu65, *Mn55, *Mg24,
*Mg25, *Mg26, *Zn64, *Zn66, *Zn67, *Zn68, *Zn70,
*Al27, *Ti46, *Ti47, *Ti48, *Ti49, *Ti50, *Na23,
*Ga69, *Ga71, *H1, *C12, *C13, *Zr90, *Zr91,
*Zr92, *Zr94, *Zr96, *Sn112, *Sn114, *Sn115, *Sn116,
*Sn117, *Sn118, *Sn119, *Sn120, *Sn122, *Sn124,
*Ca40, *Ca42, *Ca43, *Ca44, *Ca46, *Ca48, *B10, *B11,
*Li6, *Li7, *Gd152,*Gd154, *Gd155, *Gd156, *Gd157,
*Gd158, *Gd160,*V50, *V51;
G4Element *Oxygen, *Deuterium, *LEU,
*Cr, *Fe, *Si, *Cu, *Mn, *Mg, *Zn, *Al,
*Ti, *Na, *Ga, *Hydrogen, *C, *Zr, *Sn, *Ca,
```

```
*B, *Li, *Gd, *V,
*FeAl, *CuAl,*FeZr, *CrZr, *OxygenZr,
*OxygenLEU, *OxygenLW;
G4Material *World, *LEUMat,
*Aluminum57S, *AlPresT, *AlCalT, *H2O, *D2O,
*AnnulusGas, *Zr4, *Air, *Moderator, *Graphite;


// Create the world environment
World = new G4Material("Galactic", 1, 1, 1.e-25*g/cm3, kStateGas,2.73*kelvin, 3.e-
18*pascal);


//make Calcium isotopes and element
Ca40 = new G4Isotope("Ca40", 20, 40, 39.9625906*g/mole);
Ca42 = new G4Isotope("Ca42", 20, 42, 41.9586176*g/mole);
Ca43 = new G4Isotope("Ca43", 20, 43, 42.9587662*g/mole);
Ca44 = new G4Isotope("Ca44", 20, 44, 43.9554806*g/mole);
Ca46 = new G4Isotope("Ca46", 20, 46, 45.953689*g/mole);
Ca48 = new G4Isotope("Ca48", 20, 48, 47.952533*g/mole);
Ca = new G4Element("Calcium", "Ca", 6);
Ca->AddIsotope(Ca40, 96.941*perCent);
Ca->AddIsotope(Ca42, 0.647*perCent);
Ca->AddIsotope(Ca43, 0.135*perCent);
Ca->AddIsotope(Ca44, 2.086*perCent);
Ca->AddIsotope(Ca46, 0.004*perCent);
Ca->AddIsotope(Ca48, 0.187*perCent);


//make Boron isotopes and element
```

```
B10 = new G4Isotope("B10", 5, 10, 10.012937*g/mole);
B11 = new G4Isotope("B11", 5, 11, 11.009305*g/mole);
B = new G4Element("Boron", "B", 2);
B->AddIsotope(B10, 19.9*perCent);
B->AddIsotope(B11, 80.1*perCent);


//make Lithium isotopes and element
Li6 = new G4Isotope("Li6", 3, 6, 6.0151223*g/mole);
Li7 = new G4Isotope("Li7", 3, 7, 7.0160040*g/mole);
Li = new G4Element("Lithium", "Li", 2);
Li->AddIsotope(Li6, 7.59 *perCent);
Li->AddIsotope(Li7, 92.41*perCent);


//make Vanadium isotopes and element
V50 = new G4Isotope("V50", 23, 50, 49.9471609 *g/mole);
V51 = new G4Isotope("V51", 23, 51, 50.9439617 *g/mole);
V = new G4Element("Vanadium", "V", 2);
V->AddIsotope(V50, 0.250 *perCent);
V->AddIsotope(V51, 99.750*perCent);


//make chromium isotopes and element
Cr50 = new G4Isotope("Cr50", 24, 50, 49.9460422*g/mole);
Cr52 = new G4Isotope("Cr52", 24, 52, 51.9405075*g/mole);
Cr53 = new G4Isotope("Cr53", 24, 53, 52.9406494*g/mole);
Cr54 = new G4Isotope("Cr54", 24, 54, 53.9388804*g/mole);
Cr = new G4Element("Chromium", "Cr", 4);
Cr->AddIsotope(Cr50, 4.1737*perCent);
```

```
Cr->AddIsotope(Cr52, 83.7003*perCent);
Cr->AddIsotope(Cr53, 9.6726*perCent);
Cr->AddIsotope(Cr54, 2.4534*perCent);


CrZr = new G4Element("Chromium", "Cr", 4);
CrZr->AddIsotope(Cr50, 4.10399884*perCent);
CrZr->AddIsotope(Cr52, 82.20818453*perCent);
CrZr->AddIsotope(Cr53, 9.50012786*perCent);
CrZr->AddIsotope(Cr54, 4.18768878*perCent);


//make iron isotopes and element
Fe54 = new G4Isotope("Fe54", 26, 54, 53.9396105*g/mole);
Fe56 = new G4Isotope("Fe56", 26, 56, 55.9349375*g/mole);
Fe57 = new G4Isotope("Fe57", 26, 57, 56.9353940*g/mole);
Fe58 = new G4Isotope("Fe58", 26, 58, 57.9332756*g/mole);
Fe = new G4Element("Iron", "Fe", 4);
Fe->AddIsotope(Fe54, 5.80*perCent);
Fe->AddIsotope(Fe56, 91.72*perCent);
Fe->AddIsotope(Fe57, 2.20*perCent);
Fe->AddIsotope(Fe58, 0.28*perCent);
//make iron element for Aluminium material in ZED-2
FeAl = new G4Element("Iron", "Fe", 4);
FeAl->AddIsotope(Fe54, 0.02340*perCent);
FeAl->AddIsotope(Fe56, 0.36700*perCent);
FeAl->AddIsotope(Fe57, 0.00848*perCent);
FeAl->AddIsotope(Fe58, 0.00112*perCent);
//make iron element for Aluminium material in ZED-2
FeZr = new G4Element("Iron", "Fe", 4);
```

```
FeZr->AddIsotope(Fe54, 5.60198907*perCent);
FeZr->AddIsotope(Fe56, 91.9458541*perCent);
FeZr->AddIsotope(Fe57, 2.14094671*perCent);
FeZr->AddIsotope(Fe58, 0.31121012*perCent);


//make Silicon isotopes and element
Si28 = new G4Isotope("Si28", 14, 28, 27.9769271*g/mole);
Si29 = new G4Isotope("Si29", 14, 29, 28.9764949*g/mole);
Si30 = new G4Isotope("Si30", 14, 30, 29.9737707*g/mole);
Si = new G4Element("Silicon", "Si", 3);
Si->AddIsotope(Si28, 92.23*perCent);
Si->AddIsotope(Si29, 4.67*perCent);
Si->AddIsotope(Si30, 3.1*perCent);


//make Magnesium isotopes and element
Mg24 = new G4Isotope("Mg24", 12, 24, 23.9850423*g/mole);
Mg25 = new G4Isotope("Mg25", 12, 25, 24.9858374*g/mole);
Mg26 = new G4Isotope("Mg26", 12, 26, 25.9825937 *g/mole);
Mg = new G4Element("Magnesium", "Mg", 3);
Mg->AddIsotope(Mg24, 78.99*perCent);
Mg->AddIsotope(Mg25, 10.00*perCent);
Mg->AddIsotope(Mg26, 11.01*perCent);


//make Manganese isotopes and element
Mn55 = new G4Isotope("Mn55", 25, 55, 54.9380471*g/mole);
Mn = new G4Element("Manganese", "Mn", 1);
```

```
Mn->AddIsotope(Mn55, 100.00*perCent);


//make Copper isotopes and element
Cu63 = new G4Isotope("Cu63", 29, 63, 62.9295989*g/mole);
Cu65 = new G4Isotope("Cu65", 29, 65, 64.9277929 *g/mole);
Cu = new G4Element("Copper", "Cu", 2);
Cu->AddIsotope(Cu63, 69.17*perCent);
Cu->AddIsotope(Cu65, 30.83*perCent);
//make copper for Al
CuAl = new G4Element("Copper", "Cu", 2);
CuAl->AddIsotope(Cu63, 0.01383*perCent);
CuAl->AddIsotope(Cu65, 0.00617*perCent);


//make Aluminum isotopes and element
Al27 = new G4Isotope("Al27", 13, 27, 26.9815386 *g/mole);
Al = new G4Element("Aluminum", "Al", 1);
Al->AddIsotope(Al27, 100.00*perCent);


//make Zirconium isotopes and element
Zr90 = new G4Isotope("Zr90", 40, 90, 89.9047026*g/mole);
Zr91 = new G4Isotope("Zr91", 40, 91, 90.9056439*g/mole);
Zr92 = new G4Isotope("Zr92", 40, 92, 91.9050386*g/mole);
Zr94 = new G4Isotope("Zr94", 40, 94, 93.9063148*g/mole);
Zr96 = new G4Isotope("Zr96", 40, 96, 95.908275*g/mole);
Zr = new G4Element("Zirconium", "Zr", 5);
Zr->AddIsotope(Zr90, 0.5075558873*perCent);
Zr->AddIsotope(Zr91, 0.1116101232*perCent);
```

```
Zr->AddIsotope(Zr92, 0.1722780975*perCent);
Zr->AddIsotope(Zr94, 0.1791179604*perCent);
Zr->AddIsotope(Zr96, 0.0294379317*perCent);
//make Zinc isotopes and element
Zn64 = new G4Isotope("Zn64", 30, 64, 63.9291448*g/mole);
Zn66 = new G4Isotope("Zn66", 30, 66, 65.9260347*g/mole);
Zn67 = new G4Isotope("Zn67", 30, 67, 66.9271291*g/mole);
Zn68 = new G4Isotope("Zn68", 30, 68, 67.9248459*g/mole);
Zn70 = new G4Isotope("Zn70", 30, 70, 69.925325*g/mole);
Zn = new G4Element("Zinc", "Zn", 5);
Zn->AddIsotope(Zn64, 48.63*perCent);
Zn->AddIsotope(Zn66, 27.90*perCent);
Zn->AddIsotope(Zn67, 4.10*perCent);
Zn->AddIsotope(Zn68, 18.75*perCent);
Zn->AddIsotope(Zn70, 0.62*perCent);


//make Tin isotopes and element
Sn112 = new G4Isotope("Sn112", 50, 112, 111.904826*g/mole);
Sn114 = new G4Isotope("Sn114", 50, 114, 113.902784*g/mole);
Sn115 = new G4Isotope("Sn115", 50, 115, 114.903348*g/mole);
Sn116 = new G4Isotope("Sn116", 50, 116, 115.901747*g/mole);
Sn117 = new G4Isotope("Sn117", 50, 117, 116.902956*g/mole);
Sn118 = new G4Isotope("Sn118", 50, 118, 117.901609*g/mole);
Sn119 = new G4Isotope("Sn119", 50, 119, 118.903311*g/mole);
Sn120 = new G4Isotope("Sn120", 50, 120, 119.9021991*g/mole);
Sn122 = new G4Isotope("Sn122", 50, 122, 121.9034404*g/mole);
Sn124 = new G4Isotope("Sn124", 50, 124, 123.9052743*g/mole);
Sn = new G4Element("Tin", "Sn", 10);
```

```
Sn->AddIsotope(Sn112, 0.97*perCent);
Sn->AddIsotope(Sn114, 0.66*perCent);
Sn->AddIsotope(Sn115, 0.34*perCent);
Sn->AddIsotope(Sn116, 14.54*perCent);
Sn->AddIsotope(Sn117, 7.68*perCent);
Sn->AddIsotope(Sn118, 24.22*perCent);
Sn->AddIsotope(Sn119, 8.59*perCent);
Sn->AddIsotope(Sn120, 32.58*perCent);
Sn->AddIsotope(Sn122, 4.63*perCent);
Sn->AddIsotope(Sn124, 0.0*perCent);

// Soudium Isotopes
Na23 = new G4Isotope("Na23", 11, 23, 22.9897677*g/mole);
// Naturally occurring Sodium
Na = new G4Element("Soudium", "Na", 1);
Na->AddIsotope(Na23, 1.);

// Gallium Isotopes
Ga69 = new G4Isotope("Ga69", 31, 69, 68.9255809*g/mole);
Ga71 = new G4Isotope("Ga71", 31, 71, 70.9247005*g/mole);
// Naturally Occurring Gallium
Ga = new G4Element("Gallium", "Ga", 2);
Ga->AddIsotope(Ga69, 60.108*perCent);
Ga->AddIsotope(Ga71, 39.892*perCent);

//make Gadolinium isotopes and element
Gd152 = new G4Isotope("Gd152", 64, 152, 151.919786*g/mole);
```

```
Gd154 = new G4Isotope("Gd154", 64, 154, 153.920861*g/mole);
Gd155 = new G4Isotope("Gd155", 64, 155, 154.922618*g/mole);
Gd156 = new G4Isotope("Gd156", 64, 156, 155.922118*g/mole);
Gd157 = new G4Isotope("Gd157", 64, 157, 156.923956*g/mole);
Gd158 = new G4Isotope("Gd158", 64, 158, 157.924019*g/mole);
Gd160 = new G4Isotope("Gd160", 64, 160, 159.927049*g/mole);
Gd = new G4Element("Gadolinium", "Gd", 7);
Gd->AddIsotope(Gd152, 0.20*perCent);
Gd->AddIsotope(Gd154, 2.18*perCent);
Gd->AddIsotope(Gd155, 14.80*perCent);
Gd->AddIsotope(Gd156, 20.47*perCent);
Gd->AddIsotope(Gd157, 15.65*perCent);
Gd->AddIsotope(Gd158, 24.84*perCent);
Gd->AddIsotope(Gd160, 21.86*perCent);


//make titanium isotopes and element
Ti46 = new G4Isotope("Ti46", 22, 46, 45.9526294*g/mole);
Ti47 = new G4Isotope("Ti47", 22, 47, 46.9517640*g/mole);
Ti48 = new G4Isotope("Ti48", 22, 48, 47.9479473*g/mole);
Ti49 = new G4Isotope("Ti49", 22, 49, 48.9478711*g/mole);
Ti50 = new G4Isotope("Ti50", 22, 50, 49.9447921*g/mole);
Ti = new G4Element("Titanium", "Zn", 5);
Ti->AddIsotope(Ti46, 8.25*perCent);
Ti->AddIsotope(Ti47, 7.44*perCent);
Ti->AddIsotope(Ti48, 73.72*perCent);
Ti->AddIsotope(Ti49, 5.41*perCent);
Ti->AddIsotope(Ti50, 5.18*perCent);
```

```
//make Carbon isotopes and element
C12 = new G4Isotope("C12", 6, 12, 12.0*g/mole);
C13 = new G4Isotope("C13", 6, 13, 13.00335*g/mole);
C = new G4Element("Carbon", "C", 2);
C->AddIsotope(C12, 98.83*perCent);
C->AddIsotope(C13, 1.07*perCent);


// Make the uranium isotopes and element
U234 = new G4Isotope("U234", 92, 234, 234.0410*g/mole);
U235 = new G4Isotope("U235", 92, 235, 235.0439*g/mole);
U236 = new G4Isotope("U236", 92, 236, 236.0456*g/mole);
U238 = new G4Isotope("U238", 92, 238, 238.0508*g/mole);


// Make hydrogen isotopes and elements
H1 = new G4Isotope("H1", 1, 1, 1.0078*g/mole);
Hydrogen = new G4Element("Hydrogen", "H", 1);
Hydrogen->AddIsotope(H1, 100*perCent);


D2 = new G4Isotope("D2", 1, 2, 2.014*g/mole);
Deuterium = new G4Element("Deuterium", "D", 1);
Deuterium->AddIsotope(D2, 100*perCent);


// Make Oxygen isotopes and elements
O16 = new G4Isotope("O16", 8, 16, 15.9949146*g/mole);
O17 = new G4Isotope("O17", 8, 17, 16.9991312*g/mole);
```

```
Oxygen = new G4Element("Oxygen", "O", 2);
Oxygen->AddIsotope(O16, 99.963868927*perCent);
Oxygen->AddIsotope(O17, 0.036131072*perCent);


OxygenZr = new G4Element("Oxygen", "O", 1);
OxygenZr->AddIsotope(O16, 0.688463*perCent);


OxygenLEU = new G4Element("Oxygen", "O", 1);
OxygenLEU->AddIsotope(O16, 100.0*perCent);


// Making Oxygen for the light water
OxygenLW = new G4Element("OxygenLW", "OLW", 2);
OxygenLW->AddIsotope(O16, 99.995998592*perCent);
OxygenLW->AddIsotope(O17, 0.004001407*perCent);


// Making hydrogen for the lightwater
Hydrogen = new G4Element("HydrogenLW", "HLW", 1);
Hydrogen->AddIsotope(H1, 100*perCent);


LEU = new G4Element("Low Enriched Uranium","LEU",4);
LEU->AddIsotope(U234, 0.007432*perCent);
LEU->AddIsotope(U235, 0.9583*perCent);
LEU->AddIsotope(U236, 0.000239*perCent);
LEU->AddIsotope(U238, 99.0341*perCent);
```

```
// Make the LEU material
LEUMat = new G4Material("U235 Material", 10.52*g/cm3, 2,kStateSolid, 299.51*kelvin);
LEUMat->AddElement(LEU,88.146875681*perCent);
LEUMat->AddElement(OxygenLEU,11.853119788*perCent);


// Create H20 material
H2O = new G4Material("Light Water", 0.99745642056*g/cm3, 2, kStateLiquid);
H2O->AddElement(OxygenLW, 1);
H2O->AddElement(Hydrogen, 2);


D2O = new G4Material("Heavy Water", 1.10480511492*g/cm3, 2, kStateLiquid);
D2O->AddElement(Oxygen, 1);
D2O->AddElement(Deuterium, 2);


Graphite = new G4Material("Graphite", 1.64*g/cm3, 5, kStateSolid);
Graphite->AddElement(Li, 1.7e-5*perCent);
Graphite->AddElement(B, 3.e-5*perCent);
Graphite->AddElement(C, 99.99697797*perCent);
Graphite->AddElement(V, 0.00300031*perCent);
Graphite->AddElement(Gd, 2.e-5*perCent);


// Make Argon
G4Element* Ar = new G4Element("Argon", "Ar", 18., 39.948*g/mole);
// Make Argon
G4Element* N = new G4Element("Nitrogen", "N", 7., 14.01*g/mole);
```

```
//Create Aluminum57S (Reactor Calandria)
Aluminum57S = new G4Material("Aluminuum 57S", 2.7*g/cm3, 8, kStateSolid);
Aluminum57S->AddElement(Al, 96.7*perCent);
Aluminum57S->AddElement(Si, 0.25*perCent);
Aluminum57S->AddElement(Fe, 0.4*perCent);
Aluminum57S->AddElement(Cu, 0.1*perCent);
Aluminum57S->AddElement(Mn, 0.1*perCent);
Aluminum57S->AddElement(Mg, 2.2*perCent);
Aluminum57S->AddElement(Cr, 0.15*perCent);
Aluminum57S->AddElement(Zn, 0.1*perCent);

//Create AlPresT (pressure Tube)
AlPresT = new G4Material("Aluminuum 6061", 2.712631*g/cm3, 8, kStateSolid);
AlPresT->AddElement(Al, 99.1244424*perCent);
AlPresT->AddElement(Si, 0.5922414*perCent);
AlPresT->AddElement(Fe, 0.1211379*perCent);
AlPresT->AddElement(Cu, 0.0018171*perCent);
AlPresT->AddElement(Mn, 0.0383626*perCent);
AlPresT->AddElement(Cr, 0.1211405*perCent);
AlPresT->AddElement(Li, 0.00075712*perCent);
AlPresT->AddElement(B, 0.00010095*perCent);

//Create AlCalT (calandria Tube)
AlCalT = new G4Material("Aluminuum 6063", 2.684951*g/cm3, 8, kStateSolid);
AlCalT->AddElement(Al, 99.18675267*perCent);
AlCalT->AddElement(Si, 0.509640251*perCent);
AlCalT->AddElement(Fe, 0.241396625*perCent);
AlCalT->AddElement(Li, 0.00754387*perCent);
```

```
AlCalT->AddElement(B, 0.000100586*perCent);
AlCalT->AddElement(Mn, 0.041228175*perCent);
AlCalT->AddElement(Gd, 0.000010059*perCent);
AlCalT->AddElement(Ti, 0.041228175*perCent);

Moderator = new G4Material("Moderator", 1.102597*g/cm3, 2, kStateLiquid, 299.51*kelvin);
Moderator->AddMaterial(D2O, 98.705*perCent);
Moderator->AddMaterial(H2O, 1.295*perCent);

//Create Annulus Gas
AnnulusGas = new G4Material("AnnulusGas", 0.0012*g/cm3, 2, kStateGas/*, 448.72*kelvin*/);
AnnulusGas->AddElement(C,27.11*perCent);
AnnulusGas->AddElement(Oxygen,72.89*perCent);

Zr4 = new G4Material("Zircaloy-4", 6.55*g/cm3, 4, kStateSolid);
Zr4->AddElement(Oxygen, 0.12*perCent);
Zr4->AddElement(CrZr, 0.11*perCent);
Zr4->AddElement(FeZr, 0.22*perCent);
Zr4->AddElement(Zr, 99.58*perCent);

// Make Air
Air = new G4Material("Air", 1.29*mg/cm3, 5, kStateGas);
Air->AddElement(N, 74.74095914*perCent);
Air->AddElement(Oxygen, 23.49454694*perCent);
Air->AddElement(Ar, 1.274547311*perCent); Air->AddElement(Li, 0.474350981*per-
Cent);
```

```
Air->AddElement(C, 0.015595629*perCent);


// Add materials
matMap["World"] = World;
matMap["LEUMat"] = LEUMat;
matMap["Graphite"] = Graphite;
matMap["Al57S"] = Aluminum57S;
matMap["AlPresT"] = AlPresT;
matMap["AlCalT"] = AlCalT;
matMap["Zr4"] = Zr4;
matMap["Air"] = Air;
matMap["Moderator"] = Moderator;
matMap["Coolant"] = H2O;
matChanged = false;
return;
}
```

Appendix B

Quarter Core ZED2 Construction

```
/*
Source code for the Quarter core ZED2 geometry and materials
*/
#include "ZED2Constructor.hh"
// Constructor
ZED2Constructor::ZED2Constructor()
:StorkVWorldConstructor(), tankLogical1(0),
logicRodA1(0), logicRodB1(0),logicCoolant1(0),
logicPressure1(0), logicGasAnn1(0)
{


}
// Destructor
ZED2Constructor:: ZED2Constructor()
{
// Delete visualization attributes
delete vesselVisAtt;
delete tank1VisATT;
```

```
delete ModVisAtt;
delete fuelA1VisATT;
delete fuelB1VisATT;
delete sheathA1VisATT;
delete sheathB1VisATT;
delete Air1VisAtt;
delete Coolant1VisAtt;
delete Pressure1VisAtt;
delete GasAnn1VisAtt;
delete Calandria1VisAtt;
delete EndPlate2VisATT;
delete airTubeLogical;
delete DumplineAlVisAtt;
delete DumplineHWVisAtt;
}
// ConstructNewWorld()
G4VPhysicalVolume* ZED2Constructor::ConstructNewWorld(const StorkParseInput*
infile) {

// Call base class ConstructNewWorld() to complete construction
return StorkVWorldConstructor::ConstructNewWorld(infile);
}



// ConstructWorld
// Construct the geometry and materials of the reactor given the inputs.
G4VPhysicalVolume* ZED2Constructor::ConstructWorld()
{
```

```
// Set local variables and enclosed world dimensions
reactorDim = G4ThreeVector(0.*cm, 231.806*cm ,405.4*cm/2.);
G4double buffer = 1.0*cm;
encWorldDim = G4ThreeVector(2*reactorDim[1]+buffer, 2*reactorDim[1]+buffer, 2*re-
actorDim[2]+buffer);
G4SolidStore* theSolids = G4SolidStore::GetInstance();

//Defining the graphite wall and bottom
G4double Graphitewall[3] =171.806*cm, 231.806*cm, 315.4*cm/2.;
G4double Graphitebott[3] = 0., 231.806*cm,90.0*cm/2.;

// Create Dimensions of Calandria Tank
G4double CalandriaDim1[3] =  0.*cm, 168.635*cm, 315.4*cm/2.-2.69*cm/2.;
G4double BotReacTankDim[3] = 0.*cm, 168.635*cm, 2.69*cm/2.;

// Defining the dimensions of the moderator
G4double ModHeight = 132.707*cm;
G4double distbtwflrtofuel = 10.1124*cm;
G4double RodHeight = 2.0*CalandriaDim1[2]-distbtwflrtofuel;
G4double MTankDim[3] = 0.*cm, 168.0*cm, ModHeight;
G4double TubeAirFuel[3] = 0.0*cm, 168.0*cm, (2*CalandriaDim1[2]-ModHeight)/2.;

// Create Dimensions of Fuel Assembly
G4double CalendriaT1Dim[3] = 0.0*cm, 12.74*cm, RodHeight;
G4double GasAnn1Dim[3] = 0.0*cm, 12.46*cm, RodHeight;
G4double PressureT1Dim[3] = 0.0*cm, 10.78*cm, RodHeight;
```

```
G4double Coolant1Dim[3] = 0.0*cm, 10.19*cm, (5.*(49.51*cm));
G4double Air1Dim[3] = 0.0*cm, 10.19*cm, RodHeight-(5.*(49.51*cm));
G4double EndPlate2[3] =  0.0*cm, 4.585*cm, 0.16*cm/2.0;
G4double FuelRodADim1[3] = 0.0*cm, 1.264*cm,48.25*cm/2.;
G4double FuelRodBDim1[3] = 0.0*cm, 1.070*cm,48.0*cm/2.;
G4double SheathADim1[3] = 0.0*cm, 1.350*cm, 49.19*cm/2.;
G4double SheathBDim1[3] = 0.0*cm, 1.150*cm, 49.19*cm/2.;


// Create the ring for fuel pins placement
G4int rings = 4;
G4double ringRad[3] = 1.734*cm,3.075*cm,4.384*cm;
G4double secondRingOffset = 0.261799*radian;


// Calculating the fuel cuts in the moderator and air
G4double topCalandriatoModH = 2.*CalandriaDim1[2]-ModHeight;
G4double AirinCT = RodHeight-Coolant1Dim[2];
G4double topFueltoModH = topCalandriatoModH-AirinCT;
G4double FuelinModH = Coolant1Dim[2]-(topFueltoModH);
G4int ModFuelIntersectPin = floor((((topFueltoModH)/10.)/49.51*cm)/10.);
G4int NumOfFuelBunInMod = floor((((FuelinModH)/10.)/49.51*cm)/10.);
G4double FullFuelBunInAir = ModFuelIntersectPin*49.51*cm;
G4double ModFuelIntersectPos = (topFueltoModH-FullFuelBunInAir-0.16*cm);
G4double CutFuelBunInMod = 49.19*cm-ModFuelIntersectPos;


// Create Dimensions of dump lines in graphite
G4double DumpLineAlDim[3] = 0.0*cm, 22.86*cm, 90.*cm/2.;
```

```
G4double DumplineHWDim[3] = 0.0*cm, 22.066*cm, 90.*cm/2.;


// Create Dimensions of dump lines in Al calandria
G4double DumpLineAlDimC[3] = 0.0*cm, 22.86*cm, 2.69*cm/2.;
G4double DumplineHWDimC[3] = 0.0*cm, 22.066*cm, 2.69*cm/2.;


// Positions of the fuel bundles
G4double Pich[2] = 24.5*cm, 24.5*cm;


G4double XPos[] = {
Pich[0]/2,Pich[0]/2,Pich[0]/2,Pich[0]/2,
3*Pich[0]/2, 3*Pich[0]/2, 3*Pich[0]/2, 3*Pich[0]/2,
5*Pich[0]/2,5*Pich[0]/2,5*Pich[0]/2,
7*Pich[0]/2,7*Pich[0]/2};


G4double YPos[] = {
Pich[1]/2, 3.*Pich[1]/2, 5.*Pich[1]/2, 7.*Pich[1]/2,
Pich[1]/2, 3.*Pich[1]/2, 5.*Pich[1]/2, 7.*Pich[1]/2,
Pich[1]/2, 3.*Pich[1]/2, 5.*Pich[1]/2,
Pich[1]/2, 3.*Pich[1]/2
};


// Set up the materials (if necessary)
if(matChanged)
{
// Delete any existing materials
DestroyMaterials();
```

```
// Create the materials
ConstructMaterials();
}
// Clean up volumes
G4GeometryManager::GetInstance()->OpenGeometry();
G4PhysicalVolumeStore::GetInstance()->Clean();
G4LogicalVolumeStore::GetInstance()->Clean();

// Set up the solids if necessary
if(geomChanged)
{
// Clean up solids
G4SolidStore::GetInstance()->Clean();
// Clean up solids
G4SolidStore::GetInstance()->Clean();

// Create world solid
new G4Box("ZED2World", encWorldDim[0]/2., encWorldDim[1]/2. , encWorldDim[2]/2.);
// Create the air above the moderator
new G4Tubs("AirTube", TubeAirFuel[0], TubeAirFuel[1], TubeAirFuel[2], 0., CLHEP::pi/2.0);

// Create Graphite Reflector solid
new G4Tubs("graphitewall", Graphitewall[0], Graphitewall[1], Graphitewall[2], 0., CLHEP::pi/2.0);
new G4Tubs("graphitebott", Graphitewall[0], Graphitewall[1], Graphitebott[2], 0., CLHEP::pi/2.0);
new G4UnionSolid("graphitewall+graphitebott", theSolids->GetSolid("graphitewall"),
theSolids->GetSolid("graphitebott"), 0, G4ThreeVector(0.,0.,-Graphitewall[2]-Graphitebott[2]));
// Create Sheilding walls
```

```
//new G4Tubs("sheildingwall", Shieldingwall[0], Shieldingwall[1], Shieldingwall[2],
0., 2.0*CLHEP::pi);


// Create the Calandria solids 1
new G4Tubs("calandriashell", CalandriaDim1[0], CalandriaDim1[1], CalandriaDim1[2],
0., CLHEP::pi/2.0);
new G4Tubs("calandriabott", BotReacTankDim[0], BotReacTankDim[1], BotReacTankDim[2],
0., CLHEP::pi/2.0);
new G4UnionSolid("calandriashell+calandriabott", theSolids->GetSolid("calandriashell"),
theSolids->GetSolid("calandriabott"), 0, G4ThreeVector(0,0,(-CalandriaDim1[2]-BotReacTankDim[2]


// Create Moderator solid
new G4Tubs("ModSphere", MTankDim[0], MTankDim[1], MTankDim[2]/2., 0., CLHEP::pi/2.0);


// Create the air above the coolant tube solid
new G4Tubs("AirTube1", Air1Dim[0]/2, Air1Dim[1]/2, Air1Dim[2]/2., 0., 2.0*CLHEP::pi);
// Create the Calandria tube
//new G4Tubs("CalandriaTubedwn", CalendriaT1Dim[0]/2, CalendriaT1Dim[1]/2, Cal-
endriaT1Dim[2], 0., 2.0*CLHEP::pi);
new G4Tubs("CalandriaTubedwnCut1", CalendriaT1Dim[0]/2, CalendriaT1Dim[1]/2,
(CalendriaT1Dim[2]-topCalandriatoModH)/2., 0., 2.0*CLHEP::pi);
new G4Tubs("CalandriaTubedwnCut2", CalendriaT1Dim[0]/2, CalendriaT1Dim[1]/2,
topCalandriatoModH/2., 0., 2.0*CLHEP::pi);


// Create the GasAnn tube solid
//new G4Tubs("GasAnnTube1", GasAnn1Dim[0]/2, GasAnn1Dim[1]/2, GasAnn1Dim[2],
0., 2.0*CLHEP::pi);
```

```
new G4Tubs("GasAnnTube1Cut1", GasAnn1Dim[0]/2, GasAnn1Dim[1]/2, (GasAnn1Dim[2]-
topCalandriatoModH)/2., 0., 2.0*CLHEP::pi);
new G4Tubs("GasAnnTube1Cut2", GasAnn1Dim[0]/2, GasAnn1Dim[1]/2, topCalan-
driatoModH/2., 0., 2.0*CLHEP::pi);


// Create the pressure tube solid
//new G4Tubs("PressureTubedwn", PressureT1Dim[0]/2, PressureT1Dim[1]/2, PressureT1Dim[2],
0., 2.0*CLHEP::pi);
new G4Tubs("PressureTubedwnCut1", PressureT1Dim[0]/2, PressureT1Dim[1]/2, (GasAnn1Dim[2]-
topCalandriatoModH)/2, 0., 2.0*CLHEP::pi);
new G4Tubs("PressureTubedwnCut2", PressureT1Dim[0]/2, PressureT1Dim[1]/2, top-
CalandriatoModH/2., 0., 2.0*CLHEP::pi);


// Create the coolant tube solid
//new G4Tubs("CoolantTube1", Coolant1Dim[0]/2, Coolant1Dim[1]/2, Coolant1Dim[2],
0., 2.0*CLHEP::pi);
new G4Tubs("CoolantTube1Cut1", Coolant1Dim[0]/2, Coolant1Dim[1]/2, FuelinModH/2./*-
topFueltoModH+(CalendriaT1Dim[2]-Coolant1Dim[2]) */, 0., 2.0*CLHEP::pi);
new G4Tubs("CoolantTube1Cut2", Coolant1Dim[0]/2, Coolant1Dim[1]/2, topFueltoModH/2./*(Calen
Coolant1Dim[2])/2+Coolant1Dim[2]-topFueltoModH*/, 0., 2.0*CLHEP::pi);


// Create outer fuel bunndles solid
new G4Tubs("FuelTubeB1", FuelRodBDim1[0]/2, FuelRodBDim1[1]/2, FuelRodBDim1[2],
0., 2.0*CLHEP::pi);
new G4Tubs("FuelTubeB1Cut1", FuelRodBDim1[0]/2, FuelRodBDim1[1]/2, CutFuel-
BunInMod/2., 0., 2.0*CLHEP::pi);
```

```
new G4Tubs("FuelTubeB1Cut2", FuelRodBDim1[0]/2, FuelRodBDim1[1]/2, ModFu-
elIntersectPos/2., 0., 2.0*CLHEP::pi);


// Create inner fuel bunndles solid
new G4Tubs("FuelTubeA1", FuelRodADim1[0]/2, FuelRodADim1[1]/2, FuelRodADim1[2],
0., 2.0*CLHEP::pi);
new G4Tubs("FuelTubeA1Cut1", FuelRodADim1[0]/2, FuelRodADim1[1]/2, CutFu-
elBunInMod/2., 0., 2.0*CLHEP::pi);
new G4Tubs("FuelTubeA1Cut2", FuelRodADim1[0]/2, FuelRodADim1[1]/2, ModFu-
elIntersectPos/2., 0., 2.0*CLHEP::pi);


// Create outer Zr-4 sheath solid
new G4Tubs("SheathB1", SheathBDim1[0]/2, SheathBDim1[1]/2, SheathBDim1[2],
0., 2.0*CLHEP::pi);
new G4Tubs("SheathB1Cut1", SheathBDim1[0]/2, SheathBDim1[1]/2, CutFuelBunIn-
Mod/2., 0., 2.0*CLHEP::pi);
new G4Tubs("SheathB1Cut2", SheathBDim1[0]/2, SheathBDim1[1]/2, ModFuelInter-
sectPos/2., 0., 2.0*CLHEP::pi);


// Create inner Zr-4 sheath solid
new G4Tubs("SheathA1", SheathADim1[0]/2, SheathADim1[1]/2, SheathADim1[2],
0., 2.0*CLHEP::pi);
new G4Tubs("SheathA1Cut1", SheathADim1[0]/2, SheathADim1[1]/2, CutFuelBun-
InMod/2., 0., 2.0*CLHEP::pi);
new G4Tubs("SheathA1Cut2", SheathADim1[0]/2, SheathADim1[1]/2, ModFuelInter-
sectPos/2., 0., 2.0*CLHEP::pi);
```

```
// Create the end plate
new G4Tubs("EndPlate2", EndPlate2[0], EndPlate2[1], EndPlate2[2], 0., 2.0*CLHEP::pi);
new G4Tubs("EndPlate1", EndPlate2[0], EndPlate2[1], EndPlate2[2], 0., 2.0*CLHEP::pi);
// Create Dump Line solid in graphite
new G4Tubs("DumpLineAl", DumpLineAlDim[0]/2, DumpLineAlDim[1]/2, DumpLin-
eAlDim[2], 0., 2.0*CLHEP::pi);
new G4Tubs("DumpLineHW", DumplineHWDim[0]/2, DumplineHWDim[1]/2, Dumpline-
HWDim[2], 0., 2.0*CLHEP::pi);
// Create Dump Line solid in Al Calandria
new G4Tubs("DumpLineAlC", DumpLineAlDimC[0]/2, DumpLineAlDimC[1]/2, DumpLin-
eAlDimC[2], 0., 2.0*CLHEP::pi);
new G4Tubs("DumpLineHWC", DumplineHWDimC[0]/2, DumplineHWDimC[1]/2,
DumplineHWDimC[2], 0., 2.0*CLHEP::pi);

geomChanged = false;
}

// Create world volume
// Create world volume
worldLogical = new G4LogicalVolume(theSolids->GetSolid("ZED2World"),matMap["World"],
"worldLogical",0,0,0); worldPhysical = new G4PVPlacement(0, G4ThreeVector(0.,0.,0.),
worldLogical,"worldPhysical",0,0,0);

// Create Reflector volume
vesselLogical = new G4LogicalVolume(theSolids->GetSolid("graphitewall+graphitebott"),
matMap["Graphite"], "VesselLogical", 0, 0, 0);
```

```
new G4PVPlacement(0,G4ThreeVector(0,0,Graphitebott[2]), vesselLogical, "VesselPhys-
ical", worldLogical, 0, 0);

// Create Calandrai volume in mother air volume tankLogical1 = new G4LogicalVolume(theSolids-
>GetSolid("calandriashell+calandriabott"), matMap["Al57S"], "VesselLogical1", 0, 0,
0);
new G4PVPlacement(0, G4ThreeVector(0.,0.,BotReacTankDim[2]), tankLogical1,"CalandriaPhysical1
vesselLogical,0,0);

// Create Moderator volume ModLogical = new G4LogicalVolume(theSolids->GetSolid("ModSphere")
"ModLogical",0,0,0);
new G4PVPlacement(0, G4ThreeVector(0.,0.,MTankDim[2]/2.-CalandriaDim1[2]), Mod-
Logical, "ModPhysical", tankLogical1, false, 0);

//Create Air above the moderator airTubeLogical = new G4LogicalVolume(theSolids-
>GetSolid("AirTube"),matMap["Air"], "airTubeLogical",0,0,0); new G4PVPlacement(0,
G4ThreeVector(0.,0., TubeAirFuel[2]+MTankDim[2]-CalandriaDim1[2]), airTubeLog-
ical, "airTubePhysical", tankLogical1,0,0);
std::stringstream volName;

logicCalandria1 = new G4LogicalVolume(theSolids->GetSolid("CalandriaTubedwnCut2"),
matMap["AlCalT"], "CalandriaTube1LogicalCut2", 0, 0, 0);
for (G4int i=0; i<52; i++)
{
// Create calandria tubes in mother air volume
volName.str("");
volName.clear();
```

```
volName « i;
new G4PVPlacement (0, G4ThreeVector(XPos[i],YPos[i],0.), logicCalandria1, "Calan-
driaTube1PhysicalCut2"+volName.str(), airTubeLogical, false, 0);
}
// Create gas annulus tubes in mother air volume
logicGasAnn1 = new G4LogicalVolume(theSolids->GetSolid("GasAnnTube1Cut2"), matMap["Air"],
"GasAnnTube1Logical", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,0), logicGasAnn1, "GasAnnTube1PhysicalCut2",
logicCalandria1, false, 0);

// Create pressure tubes in mother air volume
logicPressure1 = new G4LogicalVolume(theSolids->GetSolid("PressureTubedwnCut2"),
matMap["AlPresT"], "PressureTube1Logical", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,0), logicPressure1, "PressureTube1PhysicalCut2",
logicGasAnn1, false, 0);

// Create lower coolant in mother air volume
logicCoolant1 = new G4LogicalVolume(theSolids->GetSolid("CoolantTube1Cut2"), matMap["Air"],
"Coolant1Logical", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,-topCalandriatoModH/2.+topFueltoModH/2.),
logicCoolant1, "Coolant1PhysicalCut2", logicPressure1, false, 0);

// Create air
logicAir1 = new G4LogicalVolume(theSolids->GetSolid("AirTube1"), matMap["Air"],
"Air1Logical", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,-topCalandriatoModH/2.+topFueltoModH+Air1Dim[2]/2
```

```
logicAir1, "Air1Physical", logicPressure1, false, 0);


// Create inner/outer FULL fuel bunndles and sheath in air volume
logicRodA1 = new G4LogicalVolume(theSolids->GetSolid("FuelTubeA1"), matMap["LEUMat"],
"FuelRodA1Logical", 0, 0, 0);
logicRodB1 = new G4LogicalVolume(theSolids->GetSolid("FuelTubeB1"), matMap["LEUMat"],
"FuelRodB1Logical", 0, 0, 0);
logicSheathA1 = new G4LogicalVolume(theSolids->GetSolid("SheathA1"), matMap["Zr4"],
"SheathA1Logical", 0, 0, 0);
logicSheathB1 = new G4LogicalVolume(theSolids->GetSolid("SheathB1"), matMap["Zr4"],
"SheathB1Logical", 0, 0, 0);
logicEndPlate1 = new G4LogicalVolume(theSolids->GetSolid("EndPlate1"), matMap["Zr4"],
"EndPlate1", 0, 0, 0);
logicEndPlate2 = new G4LogicalVolume(theSolids->GetSolid("EndPlate2"), matMap["Zr4"],
"EndPlate2", 0, 0, 0);


for (G4int l=0; l<ModFuelIntersectPin; l++)
{
// Rotation and translation of the rod and sheathe


// place the center pin in air
new G4PVPlacement(0, G4ThreeVector(0,0, (topFueltoModH/2.-(l+1)*(SheathADim1[2]+2.*EndPlat
l*(2.*EndPlate2[2]+SheathADim1[2])) ), logicSheathA1,"sheathePhysical " + volName.str(),
logicCoolant1,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodA1,"fuelPhysicalA", logicSheathA1,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodB1,"fuelPhysicalB", logicSheathB1,0,0);
```

```
for( G4int j = 1; j < rings; j++ )
{
for( G4int k = 0; k < j*6; k++ )
{
// Reset string stream
volName.str("");

volName « j « "-" « k;

if(j == 2)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffse
ringRad[j-1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffset),(topFueltoModH/2
(l+1)*(SheathADim1[2]+2.*EndPlate2[2])-l*(2.*EndPlate2[2]+SheathADim1[2])));
new G4PVPlacement(0, Tm, logicSheathB1,"sheathePhysical " +volName.str(),logicCoolant1,0,0);

}
else if (j == 1)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),(topFueltoModH/2.-(l+1)*(SheathADim1[2]+2.*
l*(2.*EndPlate2[2]+SheathADim1[2])));
new G4PVPlacement(0, Tm, logicSheathA1,"sheathePhysical " +volName.str(),logicCoolant1,0,0);
}
else
{
```

```
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),(topFueltoModH/2.-(l+1)*(SheathADim1[2]+2.*
l*(2.*EndPlate2[2]+SheathADim1[2])));
new G4PVPlacement(0, Tm, logicSheathB1,"sheathePhysical " +volName.str(),logicCoolant1,0,0);
}
}
}

// Make the end plates 1
G4ThreeVector EP1(0,0,(topFueltoModH/2.-EndPlate2[2])-l*(49.51*cm)); new G4PVPlacement(0,
EP1, logicEndPlate1,"EndPlate1Physical1",logicCoolant1,0,0);
// Make the end plates 2
G4ThreeVector EP2(0,0,(topFueltoModH/2.-EndPlate2[2])-l*(2.*EndPlate2[2])-(l+1)*(2.*SheathADi
new G4PVPlacement(0, EP2, logicEndPlate2,"EndPlate2Physical1",logicCoolant1,0,0);

}

// Create inner/outer cut fuel bunndl in air volume
logicRodA1Cut2 = new G4LogicalVolume(theSolids->GetSolid("FuelTubeA1Cut2"),
matMap["LEUMat"], "FuelRodA1LogicalCut2", 0, 0, 0);
logicRodB1Cut2 = new G4LogicalVolume(theSolids->GetSolid("FuelTubeB1Cut2"),
matMap["LEUMat"], "FuelRodB1LogicalCut2", 0, 0, 0);
logicSheathA1Cut2 = new G4LogicalVolume(theSolids->GetSolid("SheathA1Cut2"),
matMap["Zr4"], "SheathA1Logicalcut2", 0, 0, 0);
logicSheathB1Cut2 = new G4LogicalVolume(theSolids->GetSolid("SheathB1Cut2"),
matMap["Zr4"], "SheathB1LogicalCut2", 0, 0, 0);
logicEndPlate2Cut2 = new G4LogicalVolume(theSolids->GetSolid("EndPlate2"), matMap["Zr4"],
```

```
"EndPlate2Cut2", 0, 0, 0);
// Rotation and translation of the rod and sheathe


// Set name for sheathe physical volume


volName.str("");
volName « 0;


// place the center pin in air
new G4PVPlacement(0, G4ThreeVector(0,0,-topFueltoModH/2.+ModFuelIntersectPos/2.),
logicSheathA1Cut2,"sheathePhysicalCut2 " + volName.str(), logicCoolant1,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodA1Cut2,"fuelPhysicalCut2A",
logicSheathA1Cut2,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodB1Cut2,"fuelPhysicalCut2B",
logicSheathB1Cut2,0,0);


for( G4int j = 1; j < rings; j++ )
{
for( G4int k = 0; k < j*6; k++ )
{
// Reset string stream
volName.str("");


volName « j « "-" « k;
```

```
if(j == 2)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffse
ringRad[j-1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffset),-topFueltoModH/2
new G4PVPlacement(0, Tm, logicSheathB1Cut2,"sheathePhysicalCut2 " +volName.str(),logicCoolant


}
else if (j == 1)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),-topFueltoModH/2.+ModFuelIntersectPos/2.);
new G4PVPlacement(0, Tm, logicSheathA1Cut2,"sheathePhysicalCut2 " +volName.str(),logicCoolant
}
else
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),-topFueltoModH/2.+ModFuelIntersectPos/2.);
new G4PVPlacement(0, Tm, logicSheathB1Cut2,"sheathePhysicalCut2 " +volName.str(),logicCoolant
}
}
}


// Make the end plates 2
G4ThreeVector EP2(0,0,-topFueltoModH/2.+ModFuelIntersectPos+0.08*cm); new G4PVPlacement(0
EP2, logicEndPlate2Cut2,"EndPlate2Physical1Cut2",logicCoolant1,0,0);


// *********Create Calandrai volume in moderator volume************
```

```
logicCalandria1Mod = new G4LogicalVolume(theSolids->GetSolid("CalandriaTubedwnCut1"),
matMap["AlCalT"], "CalandriaTube1ModLogical", 0, 0, 0);
for (G4int i=0; i<52; i++)
{
// Create calandria tubes in moderator volume
volName.str("");
volName.clear();
volName « i;
new G4PVPlacement (0, G4ThreeVector(XPos[i],YPos[i],distbtwflrtofuel/2.), logicCa-
landria1Mod, "CalandriaTube1ModPhysicalCut1"+volName.str(), ModLogical, false,
0);
}
// Create gas annulus tubes in moderator volume
logicGasAnn1Mod = new G4LogicalVolume(theSolids->GetSolid("GasAnnTube1Cut1"),
matMap["Air"], "GasAnnTube1Logical", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,0), logicGasAnn1Mod, "GasAnnTube1PhysicalCut1",
logicCalandria1Mod, false, 0);

// Create pressure tubes in moderator volume
logicPressure1Mod = new G4LogicalVolume(theSolids->GetSolid("PressureTubedwnCut1"),
matMap["AlPresT"], "PressureTube1Logical", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,0), logicPressure1Mod, "PressureTube1PhysicalCut1",
logicGasAnn1Mod, false, 0);

// Create lower coolant in moderator volume
logicCoolant1Mod = new G4LogicalVolume(theSolids->GetSolid("CoolantTube1Cut1"),
matMap["Air"], "Coolant1Logical", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,0), logicCoolant1Mod, "Coolant1PhysicalCut1",
```

```
logicPressure1Mod, false, 0);


// Create inner/outer fuel bunndle and sheath in Moderator volume
logicRodA1Mod = new G4LogicalVolume(theSolids->GetSolid("FuelTubeA1"), matMap["LEUMat"],
"FuelRodA1LogicalMod", 0, 0, 0);
logicRodB1Mod = new G4LogicalVolume(theSolids->GetSolid("FuelTubeB1"), matMap["LEUMat"],
"FuelRodB1LogicalMod", 0, 0, 0);
logicSheathA1Mod = new G4LogicalVolume(theSolids->GetSolid("SheathA1"), matMap["Zr4"],
"SheathA1LogicalMod", 0, 0, 0);
logicSheathB1Mod = new G4LogicalVolume(theSolids->GetSolid("SheathB1"), matMap["Zr4"],
"SheathB1LogicalMod", 0, 0, 0);
logicEndPlate1Mod = new G4LogicalVolume(theSolids->GetSolid("EndPlate1"), matMap["Zr4"],
"EndPlate1Mod", 0, 0, 0);
logicEndPlate2Mod = new G4LogicalVolume(theSolids->GetSolid("EndPlate2"), matMap["Zr4"],
"EndPlate2Mod", 0, 0, 0);


for (G4int l=0; l<NumOfFuelBunInMod; l++)
{
// Rotation and translation of the rod and sheathe


// Set name for sheathe physical volume
// place the center pin
new G4PVPlacement(0, G4ThreeVector(0,0, (-FuelinModH/2.+(l+1)*(SheathADim1[2]+2.*EndPlate2
)), logicSheathA1Mod,"sheathePhysicalMod " + volName.str(), logicCoolant1Mod,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodA1Mod,"fuelPhysicalModA
", logicSheathA1Mod,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodB1Mod,"fuelPhysicalModB ",
```

```
logicSheathB1Mod,0,0);


for( G4int j = 1; j < rings; j++ )
{
for( G4int k = 0; k < j*6; k++ )
{
// Reset string stream
volName.str("");


volName « j « "-" « k;


if(j == 2)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffse
ringRad[j-1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffset),(-FuelinModH/2.+
new G4PVPlacement(0, Tm, logicSheathB1Mod,"sheathePhysicalMod " +volName.str(),logicCoolant


}
else if (j == 1)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),(-FuelinModH/2.+(l+1)*(SheathADim1[2]+2.*E
new G4PVPlacement(0, Tm, logicSheathA1Mod,"sheathePhysicalMod " +volName.str(),logicCoolant


}
else
{
```

```
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),(-FuelinModH/2.+(l+1)*(SheathADim1[2]+2.*E
new G4PVPlacement(0, Tm, logicSheathB1Mod,"sheathePhysicalMod " +volName.str(),logicCoolant

}
}
}
// Make the end plates 1 G4ThreeVector EP1(0,0,(-FuelinModH/2.+EndPlate2[2])+l*(49.51*cm));
new G4PVPlacement(0, EP1, logicEndPlate1Mod,"EndPlate1Physical1Mod ",logicCoolant1Mod,0,0);
// Make the end plates 2
G4ThreeVector EP2(0,0,(-FuelinModH/2.+EndPlate2[2])+l*(2.*EndPlate2[2])+(l+1)*(2.*SheathADim
new G4PVPlacement(0, EP2, logicEndPlate2Mod,"EndPlate2Physical1Mod ",logicCoolant1Mod,0,0);
}

// Create inner/outer cut fuel bunndle in Moderator volume
logicRodA1Cut1 = new G4LogicalVolume(theSolids->GetSolid("FuelTubeA1Cut1"),
matMap["LEUMat"], "FuelRodA1LogicalCut1", 0, 0, 0);
logicRodB1Cut1 = new G4LogicalVolume(theSolids->GetSolid("FuelTubeB1Cut1"),
matMap["LEUMat"], "FuelRodB1LogicalCut1", 0, 0, 0);
logicSheathA1Cut1 = new G4LogicalVolume(theSolids->GetSolid("SheathA1Cut1"),
matMap["Zr4"], "SheathA1LogicalCut1", 0, 0, 0);
logicSheathB1Cut1 = new G4LogicalVolume(theSolids->GetSolid("SheathB1Cut1"),
matMap["Zr4"], "SheathB1LogicalCut1", 0, 0, 0);
logicEndPlate2Cut1 = new G4LogicalVolume(theSolids->GetSolid("EndPlate2"), matMap["Zr4"],
"EndPlate2Cut1", 0, 0, 0);

// place the center pin for the cut fuel bundle in the moderator
```

```
new G4PVPlacement(0, G4ThreeVector(0,0, (-FuelinModH/2.+(NumOfFuelBunInMod*49.51*cm)+2
logicSheathA1Cut1,"sheathePhysicalCut1 " + volName.str(), logicCoolant1Mod,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodA1Cut1,"fuelPhysicalCut1A
", logicSheathA1Cut1,0,0);
new G4PVPlacement(0, G4ThreeVector(0,0,0), logicRodB1Cut1,"fuelPhysicalCut1B
", logicSheathB1Cut1,0,0);


for( G4int j = 1; j < rings; j++ )
{
for( G4int k = 0; k < j*6; k++ )
{
// Reset string stream
volName.str("");


volName « j « "-" « k;


if(j == 2)
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffse
ringRad[j-1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)+secondRingOffset),(-FuelinModH/2.+
));
// place the fuel for the cut fuel bundle in the moderator
new G4PVPlacement(0, Tm, logicSheathB1Cut1,"sheathePhysicalCut1 " +volName.str(),logicCoolant


}
else if (j == 1)
{
```

```
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),(-FuelinModH/2.+(NumOfFuelBunInMod*49.5
));
new G4PVPlacement(0, Tm, logicSheathA1Cut1,"sheathePhysicalCut1 " +volName.str(),logicCoolant
}
else
{
G4ThreeVector Tm(ringRad[j-1]*cos(2.0*CLHEP::pi*G4double(k)/G4double(j*6)), ringRad[j-
1]*sin(2.0*CLHEP::pi*G4double(k)/G4double(j*6)),(-FuelinModH/2.+(NumOfFuelBunInMod*49.5
));
// place the fuel for the cut fuel bundle in the moderator
new G4PVPlacement(0, Tm, logicSheathB1Cut1,"sheathePhysicalCut1 " +volName.str(),logicCoolant
}
}
}


// Make the end plates 2
G4ThreeVector EP(0,0,(-FuelinModH/2.+(NumOfFuelBunInMod*49.51*cm)+0.08*cm));
new G4PVPlacement(0, EP, logicEndPlate2Cut1,"EndPlate2Physical1Cut1 ",logicCoolant1Mod,0,0);


// Create Dump Lines Al & Heavy Water in graphite
logicDumplineAl = new G4LogicalVolume(theSolids->GetSolid("DumpLineAl"), matMap["Al57S"],
"DumplineLogicalAl", 0, 0, 0);
for (G4int s=0; s<3; s++)
{
G4ThreeVector jg(129.54*cm*cos(2.0*CLHEP::pi*s*(2./3.)),129.54*cm*sin(2.0*CLHEP::pi*s*(2./3
Graphitewall[2]-2.*Graphitebott[2]+DumpLineAlDim[2]);
new G4PVPlacement (0, jg , logicDumplineAl, "DumLineAlPhysical", vesselLogical,
```

```
false, 0);
}
logicDumplineHW = new G4LogicalVolume(theSolids->GetSolid("DumpLineHW"),
matMap["Moderator"], "DumplineLogicalHW", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,0) , logicDumplineHW, "DumLineHW-
Physical", logicDumplineAl, false, 0);

// Create Dump Lines Al & Heavy Water in Al calandria
logicDumplineAlC = new G4LogicalVolume(theSolids->GetSolid("DumpLineAlC"),
matMap["Al57S"], "DumplineLogicalAlC", 0, 0, 0);
for (G4int d=0; d<3; d++)
{
//G4ThreeVector
lm(129.54*cm*cos(2.0*CLHEP::pi*d*(2./3.)),129.54*cm*sin(2.0*CLHEP::pi*d*(2./3.)),-
CalandriaDim1[2]-2.*BotReacTankDim[2]+DumpLineAlDimC);
G4ThreeVector jg(129.54*cm*cos(2.0*CLHEP::pi*d*(2./3.)),129.54*cm*sin(2.0*CLHEP::pi*d*(2./3
CalandriaDim1[2]-2.*BotReacTankDim[2]+DumpLineAlDimC[2]);
new G4PVPlacement (0, jg, logicDumplineAlC, "DumLineAlCPhysical",tankLogical1
, false, 0);
}

logicDumplineHWC = new G4LogicalVolume(theSolids->GetSolid("DumpLineHWC"),
matMap["Moderator"], "DumplineLogicalHWC", 0, 0, 0);
new G4PVPlacement (0, G4ThreeVector(0,0,0) , logicDumplineHWC, "DumLineHWC-
Physical", logicDumplineAlC, false, 0);

// Set reactor as sensitive detector
```

```
worldLogical->SetSensitiveDetector( sDReactor );
airTubeLogical->SetSensitiveDetector( sDReactor );
vesselLogical->SetSensitiveDetector( sDReactor );
tankLogical1->SetSensitiveDetector( sDReactor );
ModLogical->SetSensitiveDetector( sDReactor );
logicCalandria1->SetSensitiveDetector( sDReactor );
logicGasAnn1->SetSensitiveDetector( sDReactor );
logicPressure1->SetSensitiveDetector( sDReactor );
logicCoolant1->SetSensitiveDetector( sDReactor );
logicAir1->SetSensitiveDetector( sDReactor );
logicRodA1->SetSensitiveDetector( sDReactor );
logicRodB1->SetSensitiveDetector( sDReactor );
logicSheathA1->SetSensitiveDetector( sDReactor );
logicSheathB1->SetSensitiveDetector( sDReactor );
logicEndPlate2->SetSensitiveDetector( sDReactor );
logicEndPlate1->SetSensitiveDetector( sDReactor );
logicCalandria1Mod->SetSensitiveDetector( sDReactor );
logicGasAnn1Mod->SetSensitiveDetector( sDReactor );
logicPressure1Mod->SetSensitiveDetector( sDReactor );
logicCoolant1Mod->SetSensitiveDetector( sDReactor );
logicRodA1Cut2->SetSensitiveDetector( sDReactor );
logicRodB1Cut2->SetSensitiveDetector( sDReactor );
logicSheathA1Cut2->SetSensitiveDetector( sDReactor );
logicSheathB1Cut2->SetSensitiveDetector( sDReactor );
logicEndPlate2Cut2->SetSensitiveDetector( sDReactor );
logicRodA1Mod->SetSensitiveDetector( sDReactor );
logicRodB1Mod->SetSensitiveDetector( sDReactor );
logicSheathA1Mod->SetSensitiveDetector( sDReactor );
logicSheathB1Mod->SetSensitiveDetector( sDReactor );
```

```
logicEndPlate2Mod->SetSensitiveDetector( sDReactor );
logicEndPlate1Mod->SetSensitiveDetector( sDReactor );
logicRodA1Cut1->SetSensitiveDetector( sDReactor );
logicRodB1Cut1->SetSensitiveDetector( sDReactor );
logicSheathA1Cut1->SetSensitiveDetector( sDReactor );
logicSheathB1Cut1->SetSensitiveDetector( sDReactor );
logicEndPlate2Cut1->SetSensitiveDetector( sDReactor );
logicDumplineAl->SetSensitiveDetector( sDReactor );
logicDumplineHW->SetSensitiveDetector( sDReactor );
logicDumplineAlC->SetSensitiveDetector( sDReactor );
logicDumplineHWC->SetSensitiveDetector( sDReactor );

// Set visualization attributes

worldVisAtt = new G4VisAttributes(G4Colour(0.5, 1., 0.5));
worldVisAtt->SetVisibility(true);
worldLogical->SetVisAttributes(worldVisAtt);

airTubeVisAtt = new G4VisAttributes(G4Colour(0., 1., 0.5));
airTubeVisAtt->SetVisibility(true);
airTubeLogical->SetVisAttributes(airTubeVisAtt);

vesselVisAtt= new G4VisAttributes(G4Colour(1.0,0.0,0.0));
//vesselVisAtt->SetForceSolid(true);
vesselVisAtt->SetVisibility(true);
vesselLogical->SetVisAttributes(vesselVisAtt);
```

```
tank1VisATT= new G4VisAttributes(G4Colour(1.0,1.0,0.0));
//tank1VisATT->SetForceSolid(true);
tank1VisATT->SetVisibility(true);
tankLogical1->SetVisAttributes(tank1VisATT);


ModVisAtt = new G4VisAttributes(G4Colour(0.,1.,0.));
ModVisAtt->SetVisibility(true);
//ModVisAtt->SetForceSolid(true);
ModLogical->SetVisAttributes(ModVisAtt);


Calandria1VisAtt = new G4VisAttributes(G4Colour(1., 0., 1.));
//Calandria1VisAtt->SetForceSolid(true);
Calandria1VisAtt->SetVisibility(false);
logicCalandria1->SetVisAttributes(Calandria1VisAtt);
logicCalandria1Mod->SetVisAttributes(Calandria1VisAtt);


GasAnn1VisAtt = new G4VisAttributes(G4Colour(1., 0., 0.));
// GasAnn1VisAtt->SetForceSolid(true);
GasAnn1VisAtt->SetVisibility(false);
logicGasAnn1->SetVisAttributes(GasAnn1VisAtt);
logicGasAnn1Mod->SetVisAttributes(GasAnn1VisAtt);


Pressure1VisAtt = new G4VisAttributes(G4Colour(0., 1., 0.));
// Pressure1VisAtt->SetForceSolid(true);
Pressure1VisAtt->SetVisibility(false);
logicPressure1->SetVisAttributes(Pressure1VisAtt);
```

```
logicPressure1Mod->SetVisAttributes(Pressure1VisAtt);


Coolant1VisAtt = new G4VisAttributes(G4Colour(0.53, 0.81, 0.92));
//Coolant1VisAtt->SetForceSolid(true);
Coolant1VisAtt->SetVisibility(true);
logicCoolant1->SetVisAttributes(Coolant1VisAtt);
logicCoolant1Mod->SetVisAttributes(Coolant1VisAtt);


Air1VisAtt = new G4VisAttributes(G4Colour(0., 0.5, 1.));
//Air1VisAtt->SetForceSolid(true);
Air1VisAtt->SetVisibility(true);
logicAir1->SetVisAttributes(Air1VisAtt);


fuelA1VisATT = new G4VisAttributes(G4Colour(0.0, 0.0 ,1.0));
fuelA1VisATT->SetForceSolid(true);
fuelA1VisATT->SetVisibility(true);
logicRodA1->SetVisAttributes(fuelA1VisATT);
logicRodA1Cut2->SetVisAttributes(fuelA1VisATT);
logicRodA1Mod->SetVisAttributes(fuelA1VisATT);
logicRodA1Cut1->SetVisAttributes(fuelA1VisATT);


fuelB1VisATT = new G4VisAttributes(G4Colour(0,0.5,0.92));
fuelB1VisATT->SetForceSolid(true);
fuelB1VisATT->SetVisibility(true);
logicRodB1->SetVisAttributes(fuelB1VisATT);
logicRodB1Cut2->SetVisAttributes(fuelB1VisATT);
logicRodB1Mod->SetVisAttributes(fuelB1VisATT);
```

```
logicRodB1Cut1->SetVisAttributes(fuelB1VisATT);


sheathA1VisATT = new G4VisAttributes(G4Colour(0.5, 0.0 ,1.0));
//sheathA1VisATT->SetForceSolid(true);
sheathA1VisATT->SetVisibility(true);
logicSheathA1->SetVisAttributes(sheathA1VisATT);
logicSheathA1Cut2->SetVisAttributes(sheathA1VisATT);
logicSheathA1Mod->SetVisAttributes(sheathA1VisATT);
logicSheathA1Cut1->SetVisAttributes(sheathA1VisATT);


sheathB1VisATT = new G4VisAttributes(G4Colour(1.0, 0.5 ,1.0));
//sheathB1VisATT->SetForceSolid(true);
sheathB1VisATT->SetVisibility(true);
logicSheathB1->SetVisAttributes(sheathB1VisATT);
logicSheathB1Cut2->SetVisAttributes(sheathB1VisATT);
logicSheathB1Mod->SetVisAttributes(sheathB1VisATT);
logicSheathB1Cut1->SetVisAttributes(sheathB1VisATT);


EndPlate2VisATT = new G4VisAttributes(G4Colour(0.5, 0.5, 0.5));
EndPlate2VisATT->SetForceSolid(true);
EndPlate2VisATT->SetVisibility(true);
logicEndPlate2->SetVisAttributes(EndPlate2VisATT);
logicEndPlate1->SetVisAttributes(EndPlate2VisATT);
logicEndPlate2Cut2->SetVisAttributes(EndPlate2VisATT);
logicEndPlate2Mod->SetVisAttributes(EndPlate2VisATT);
logicEndPlate1Mod->SetVisAttributes(EndPlate2VisATT);
```

```
logicEndPlate2Cut1->SetVisAttributes(EndPlate2VisATT);


DumplineAlVisAtt = new G4VisAttributes(G4Colour(1., 0.99, 0.5));
DumplineAlVisAtt->SetForceSolid(false);
DumplineAlVisAtt->SetVisibility(true);
logicDumplineAl->SetVisAttributes(DumplineAlVisAtt);
logicDumplineAlC->SetVisAttributes(DumplineAlVisAtt);


DumplineHWVisAtt = new G4VisAttributes(G4Colour(0., 1.0, 0.));
DumplineHWVisAtt->SetForceSolid(false);
DumplineHWVisAtt->SetVisibility(true);
logicDumplineHW->SetVisAttributes(DumplineHWVisAtt);
logicDumplineHWC->SetVisAttributes(DumplineHWVisAtt);


return worldPhysical;
}


// ConstructMaterials()
// Construct all the materials needed for the ZED2Constructor.
void ZED2Constructor::ConstructMaterials()
{
// Elements, isotopes and materials
G4Isotope *U234, *U235, *U238, *U236, *D2, *O16, *O17,
*Fe54, *Fe56, *Fe57, *Fe58, *Cr50, *Cr52, *Cr53, *Cr54,
*Si28, *Si29, *Si30, *Cu63, *Cu65, *Mn55, *Mg24,
*Mg25, *Mg26, *Zn64, *Zn66, *Zn67, *Zn68, *Zn70,
*Al27, *Ti46, *Ti47, *Ti48, *Ti49, *Ti50, *Na23,
```

```
*Ga69, *Ga71, *H1, *C12, *C13, *Zr90, *Zr91,
*Zr92, *Zr94, *Zr96, *Sn112, *Sn114, *Sn115, *Sn116,
*Sn117, *Sn118, *Sn119, *Sn120, *Sn122, *Sn124,
*Ca40, *Ca42, *Ca43, *Ca44, *Ca46, *Ca48, *B10, *B11,
*Li6, *Li7, *Gd152,*Gd154, *Gd155, *Gd156, *Gd157,
*Gd158, *Gd160,*V50, *V51;
G4Element *Oxygen, *Deuterium, *LEU,
*Cr, *Fe, *Si, *Cu, *Mn, *Mg, *Zn, *Al,
*Ti, *Na, *Ga, *Hydrogen, *C, *Zr, *Sn, *Ca,
*B, *Li, *Gd, *V,
*FeAl, *CuAl,*FeZr, *CrZr, *OxygenZr,
*OxygenLEU, *OxygenLW;
G4Material *World, *LEUMat,
*Aluminum57S, *AlPresT, *AlCalT, *H2O, *D2O,
*AnnulusGas, *Zr4, *Air, *Moderator, *Graphite;


// Create the world environment
World = new G4Material("Galactic", 1, 1, 1.e-25*g/cm3, kStateGas,2.73*kelvin, 3.e-
18*pascal);


//make Calcium isotopes and element
Ca40 = new G4Isotope("Ca40", 20, 40, 39.9625906*g/mole);
Ca42 = new G4Isotope("Ca42", 20, 42, 41.9586176*g/mole);
Ca43 = new G4Isotope("Ca43", 20, 43, 42.9587662*g/mole);
Ca44 = new G4Isotope("Ca44", 20, 44, 43.9554806*g/mole);
Ca46 = new G4Isotope("Ca46", 20, 46, 45.953689*g/mole);
Ca48 = new G4Isotope("Ca48", 20, 48, 47.952533*g/mole);
```

```
Ca = new G4Element("Calcium", "Ca", 6);
Ca->AddIsotope(Ca40, 96.941*perCent);
Ca->AddIsotope(Ca42, 0.647*perCent);
Ca->AddIsotope(Ca43, 0.135*perCent);
Ca->AddIsotope(Ca44, 2.086*perCent);
Ca->AddIsotope(Ca46, 0.004*perCent);
Ca->AddIsotope(Ca48, 0.187*perCent);

//make Boron isotopes and element
B10 = new G4Isotope("B10", 5, 10, 10.012937*g/mole);
B11 = new G4Isotope("B11", 5, 11, 11.009305*g/mole);

B = new G4Element("Boron", "B", 2);
B->AddIsotope(B10, 19.9*perCent);
B->AddIsotope(B11, 80.1*perCent);

//make Lithium isotopes and element
Li6 = new G4Isotope("Li6", 3, 6, 6.0151223*g/mole);
Li7 = new G4Isotope("Li7", 3, 7, 7.0160040*g/mole);

Li = new G4Element("Lithium", "Li", 2);
Li->AddIsotope(Li6, 7.59 *perCent);
Li->AddIsotope(Li7, 92.41*perCent);

//make Vanadium isotopes and element
V50 = new G4Isotope("V50", 23, 50, 49.9471609 *g/mole);
```

```
V51 = new G4Isotope("V51", 23, 51, 50.9439617 *g/mole);


V = new G4Element("Vanadium", "V", 2);
V->AddIsotope(V50, 0.250 *perCent);
V->AddIsotope(V51, 99.750*perCent);


//make chromium isotopes and element
Cr50 = new G4Isotope("Cr50", 24, 50, 49.9460422*g/mole);
Cr52 = new G4Isotope("Cr52", 24, 52, 51.9405075*g/mole);
Cr53 = new G4Isotope("Cr53", 24, 53, 52.9406494*g/mole);
Cr54 = new G4Isotope("Cr54", 24, 54, 53.9388804*g/mole);


Cr = new G4Element("Chromium", "Cr", 4);
Cr->AddIsotope(Cr50, 4.1737*perCent);
Cr->AddIsotope(Cr52, 83.7003*perCent);
Cr->AddIsotope(Cr53, 9.6726*perCent);
Cr->AddIsotope(Cr54, 2.4534*perCent);


CrZr = new G4Element("Chromium", "Cr", 4);
CrZr->AddIsotope(Cr50, 4.10399884*perCent);
CrZr->AddIsotope(Cr52, 82.20818453*perCent);
CrZr->AddIsotope(Cr53, 9.50012786*perCent);
CrZr->AddIsotope(Cr54, 4.18768878*perCent);


//make iron isotopes and element
Fe54 = new G4Isotope("Fe54", 26, 54, 53.9396105*g/mole);
Fe56 = new G4Isotope("Fe56", 26, 56, 55.9349375*g/mole);
```

```
Fe57 = new G4Isotope("Fe57", 26, 57, 56.9353940*g/mole);
Fe58 = new G4Isotope("Fe58", 26, 58, 57.9332756*g/mole);


Fe = new G4Element("Iron", "Fe", 4);
Fe->AddIsotope(Fe54, 5.80*perCent);
Fe->AddIsotope(Fe56, 91.72*perCent);
Fe->AddIsotope(Fe57, 2.20*perCent);
Fe->AddIsotope(Fe58, 0.28*perCent);


//make iron element for Aluminium material in ZED-2
FeAl = new G4Element("Iron", "Fe", 4);
FeAl->AddIsotope(Fe54, 0.02340*perCent);
FeAl->AddIsotope(Fe56, 0.36700*perCent);
FeAl->AddIsotope(Fe57, 0.00848*perCent);
FeAl->AddIsotope(Fe58, 0.00112*perCent);


//make iron element for Aluminium material in ZED-2
FeZr = new G4Element("Iron", "Fe", 4);
FeZr->AddIsotope(Fe54, 5.60198907*perCent);
FeZr->AddIsotope(Fe56, 91.9458541*perCent);
FeZr->AddIsotope(Fe57, 2.14094671*perCent);
FeZr->AddIsotope(Fe58, 0.31121012*perCent);


//make Silicon isotopes and element
Si28 = new G4Isotope("Si28", 14, 28, 27.9769271*g/mole);
Si29 = new G4Isotope("Si29", 14, 29, 28.9764949*g/mole);
```

```
Si30 = new G4Isotope("Si30", 14, 30, 29.9737707*g/mole);


Si = new G4Element("Silicon", "Si", 3);
Si->AddIsotope(Si28, 92.23*perCent);
Si->AddIsotope(Si29, 4.67*perCent);
Si->AddIsotope(Si30, 3.1*perCent);


//make Magnesium isotopes and element
Mg24 = new G4Isotope("Mg24", 12, 24, 23.9850423*g/mole);
Mg25 = new G4Isotope("Mg25", 12, 25, 24.9858374*g/mole);
Mg26 = new G4Isotope("Mg26", 12, 26, 25.9825937 *g/mole);


Mg = new G4Element("Magnesium", "Mg", 3);
Mg->AddIsotope(Mg24, 78.99*perCent);
Mg->AddIsotope(Mg25, 10.00*perCent);
Mg->AddIsotope(Mg26, 11.01*perCent);


//make Manganese isotopes and element
Mn55 = new G4Isotope("Mn55", 25, 55, 54.9380471*g/mole);


Mn = new G4Element("Manganese", "Mn", 1);
Mn->AddIsotope(Mn55, 100.00*perCent);


//make Copper isotopes and element
Cu63 = new G4Isotope("Cu63", 29, 63, 62.9295989*g/mole);
```

```
Cu65 = new G4Isotope("Cu65", 29, 65, 64.9277929 *g/mole);


Cu = new G4Element("Copper", "Cu", 2);
Cu->AddIsotope(Cu63, 69.17*perCent);
Cu->AddIsotope(Cu65, 30.83*perCent);


//make copper for Al
CuAl = new G4Element("Copper", "Cu", 2);
CuAl->AddIsotope(Cu63, 0.01383*perCent);
CuAl->AddIsotope(Cu65, 0.00617*perCent);


//make Aluminum isotopes and element
Al27 = new G4Isotope("Al27", 13, 27, 26.9815386 *g/mole);


Al = new G4Element("Aluminum", "Al", 1);
Al->AddIsotope(Al27, 100.00*perCent);


//make Zirconium isotopes and element
Zr90 = new G4Isotope("Zr90", 40, 90, 89.9047026*g/mole);
Zr91 = new G4Isotope("Zr91", 40, 91, 90.9056439*g/mole);
Zr92 = new G4Isotope("Zr92", 40, 92, 91.9050386*g/mole);
Zr94 = new G4Isotope("Zr94", 40, 94, 93.9063148*g/mole);
Zr96 = new G4Isotope("Zr96", 40, 96, 95.908275*g/mole);


Zr = new G4Element("Zirconium", "Zr", 5);
Zr->AddIsotope(Zr90, 0.5075558873*perCent);
```

```
Zr->AddIsotope(Zr91, 0.1116101232*perCent);
Zr->AddIsotope(Zr92, 0.1722780975*perCent);
Zr->AddIsotope(Zr94, 0.1791179604*perCent);
Zr->AddIsotope(Zr96, 0.0294379317*perCent);

//make Zinc isotopes and element
Zn64 = new G4Isotope("Zn64", 30, 64, 63.9291448*g/mole);
Zn66 = new G4Isotope("Zn66", 30, 66, 65.9260347*g/mole);
Zn67 = new G4Isotope("Zn67", 30, 67, 66.9271291*g/mole);
Zn68 = new G4Isotope("Zn68", 30, 68, 67.9248459*g/mole);
Zn70 = new G4Isotope("Zn70", 30, 70, 69.925325*g/mole);

Zn = new G4Element("Zinc", "Zn", 5);
Zn->AddIsotope(Zn64, 48.63*perCent);
Zn->AddIsotope(Zn66, 27.90*perCent);
Zn->AddIsotope(Zn67, 4.10*perCent);
Zn->AddIsotope(Zn68, 18.75*perCent);
Zn->AddIsotope(Zn70, 0.62*perCent);

//make Tin isotopes and element
Sn112 = new G4Isotope("Sn112", 50, 112, 111.904826*g/mole);
Sn114 = new G4Isotope("Sn114", 50, 114, 113.902784*g/mole);
Sn115 = new G4Isotope("Sn115", 50, 115, 114.903348*g/mole);
Sn116 = new G4Isotope("Sn116", 50, 116, 115.901747*g/mole);
Sn117 = new G4Isotope("Sn117", 50, 117, 116.902956*g/mole);
Sn118 = new G4Isotope("Sn118", 50, 118, 117.901609*g/mole);
Sn119 = new G4Isotope("Sn119", 50, 119, 118.903311*g/mole);
```

```
Sn120 = new G4Isotope("Sn120", 50, 120, 119.9021991*g/mole);
Sn122 = new G4Isotope("Sn122", 50, 122, 121.9034404*g/mole);
Sn124 = new G4Isotope("Sn124", 50, 124, 123.9052743*g/mole);


Sn = new G4Element("Tin", "Sn", 10);
Sn->AddIsotope(Sn112, 0.97*perCent);
Sn->AddIsotope(Sn114, 0.66*perCent);
Sn->AddIsotope(Sn115, 0.34*perCent);
Sn->AddIsotope(Sn116, 14.54*perCent);
Sn->AddIsotope(Sn117, 7.68*perCent);
Sn->AddIsotope(Sn118, 24.22*perCent);
Sn->AddIsotope(Sn119, 8.59*perCent);
Sn->AddIsotope(Sn120, 32.58*perCent);
Sn->AddIsotope(Sn122, 4.63*perCent);
Sn->AddIsotope(Sn124, 0.0*perCent);


// Soudium Isotopes
Na23 = new G4Isotope("Na23", 11, 23, 22.9897677*g/mole);


// Naturally occurring Sodium
Na = new G4Element("Soudium", "Na", 1);
Na->AddIsotope(Na23, 1.);


// Gallium Isotopes
Ga69 = new G4Isotope("Ga69", 31, 69, 68.9255809*g/mole);
Ga71 = new G4Isotope("Ga71", 31, 71, 70.9247005*g/mole);
```

```
// Naturally Occurring Gallium
Ga = new G4Element("Gallium", "Ga", 2);
Ga->AddIsotope(Ga69, 60.108*perCent);
Ga->AddIsotope(Ga71, 39.892*perCent);


//make Gadolinium isotopes and element
Gd152 = new G4Isotope("Gd152", 64, 152, 151.919786*g/mole);
Gd154 = new G4Isotope("Gd154", 64, 154, 153.920861*g/mole);
Gd155 = new G4Isotope("Gd155", 64, 155, 154.922618*g/mole);
Gd156 = new G4Isotope("Gd156", 64, 156, 155.922118*g/mole);
Gd157 = new G4Isotope("Gd157", 64, 157, 156.923956*g/mole);
Gd158 = new G4Isotope("Gd158", 64, 158, 157.924019*g/mole);
Gd160 = new G4Isotope("Gd160", 64, 160, 159.927049*g/mole);


Gd = new G4Element("Gadolinium", "Gd", 7);
Gd->AddIsotope(Gd152, 0.20*perCent);
Gd->AddIsotope(Gd154, 2.18*perCent);
Gd->AddIsotope(Gd155, 14.80*perCent);
Gd->AddIsotope(Gd156, 20.47*perCent);
Gd->AddIsotope(Gd157, 15.65*perCent);
Gd->AddIsotope(Gd158, 24.84*perCent);
Gd->AddIsotope(Gd160, 21.86*perCent);


//make titanium isotopes and element
Ti46 = new G4Isotope("Ti46", 22, 46, 45.9526294*g/mole);
Ti47 = new G4Isotope("Ti47", 22, 47, 46.9517640*g/mole);
Ti48 = new G4Isotope("Ti48", 22, 48, 47.9479473*g/mole);
```

```
Ti49 = new G4Isotope("Ti49", 22, 49, 48.9478711*g/mole);
Ti50 = new G4Isotope("Ti50", 22, 50, 49.9447921*g/mole);


Ti = new G4Element("Titanium", "Zn", 5);
Ti->AddIsotope(Ti46, 8.25*perCent);
Ti->AddIsotope(Ti47, 7.44*perCent);
Ti->AddIsotope(Ti48, 73.72*perCent);
Ti->AddIsotope(Ti49, 5.41*perCent);
Ti->AddIsotope(Ti50, 5.18*perCent);


//make Carbon isotopes and element
C12 = new G4Isotope("C12", 6, 12, 12.0*g/mole);
C13 = new G4Isotope("C13", 6, 13, 13.00335*g/mole);


C = new G4Element("Carbon", "C", 2);
C->AddIsotope(C12, 98.83*perCent);
C->AddIsotope(C13, 1.07*perCent);


// Make the uranium isotopes and element
U234 = new G4Isotope("U234", 92, 234, 234.0410*g/mole);
U235 = new G4Isotope("U235", 92, 235, 235.0439*g/mole);
U236 = new G4Isotope("U236", 92, 236, 236.0456*g/mole);
U238 = new G4Isotope("U238", 92, 238, 238.0508*g/mole);


// Make hydrogen isotopes and elements
H1 = new G4Isotope("H1", 1, 1, 1.0078*g/mole);
Hydrogen = new G4Element("Hydrogen", "H", 1);
```

```
Hydrogen->AddIsotope(H1, 100*perCent);


D2 = new G4Isotope("D2", 1, 2, 2.014*g/mole);
Deuterium = new G4Element("Deuterium", "D", 1);
Deuterium->AddIsotope(D2, 100*perCent);


// Make Oxygen isotopes and elements
O16 = new G4Isotope("O16", 8, 16, 15.9949146*g/mole);
O17 = new G4Isotope("O17", 8, 17, 16.9991312*g/mole);


Oxygen = new G4Element("Oxygen", "O", 2);
Oxygen->AddIsotope(O16, 99.963868927*perCent);
Oxygen->AddIsotope(O17, 0.036131072*perCent);


OxygenZr = new G4Element("Oxygen", "O", 1);
OxygenZr->AddIsotope(O16, 0.688463*perCent);


OxygenLEU = new G4Element("Oxygen", "O", 1);
OxygenLEU->AddIsotope(O16, 100.0*perCent);


// Making Oxygen for the light water
OxygenLW = new G4Element("OxygenLW", "OLW", 2);
OxygenLW->AddIsotope(O16, 99.995998592*perCent);
OxygenLW->AddIsotope(O17, 0.004001407*perCent);
```

```
// Making hydrogen for the lightwater
Hydrogen = new G4Element("HydrogenLW", "HLW", 1);
Hydrogen->AddIsotope(H1, 100*perCent);


LEU = new G4Element("Low Enriched Uranium","LEU",4);
LEU->AddIsotope(U234, 0.007432*perCent);
LEU->AddIsotope(U235, 0.9583*perCent);
LEU->AddIsotope(U236, 0.000239*perCent);
LEU->AddIsotope(U238, 99.0341*perCent);


// Make the LEU material
LEUMat = new G4Material("U235 Material", 10.52*g/cm3, 2,kStateSolid, 299.51*kelvin);
LEUMat->AddElement(LEU,88.146875681*perCent);
LEUMat->AddElement(OxygenLEU,11.853119788*perCent);


// Create H20 material
H2O = new G4Material("Light Water", 0.99745642056*g/cm3, 2, kStateLiquid);
H2O->AddElement(OxygenLW, 1);
H2O->AddElement(Hydrogen, 2);


D2O = new G4Material("Heavy Water", 1.10480511492*g/cm3, 2, kStateLiquid);
D2O->AddElement(Oxygen, 1);
D2O->AddElement(Deuterium, 2);


Graphite = new G4Material("Graphite", 1.64*g/cm3, 5, kStateSolid);
Graphite->AddElement(Li, 1.7e-5*perCent);
Graphite->AddElement(B, 3.e-5*perCent);
```

```
Graphite->AddElement(C, 99.99697797*perCent);
Graphite->AddElement(V, 0.00300031*perCent);
Graphite->AddElement(Gd, 2.e-5*perCent);

// Make Argon
G4Element* Ar = new G4Element("Argon", "Ar", 18., 39.948*g/mole);
// Make Argon
G4Element* N = new G4Element("Nitrogen", "N", 7., 14.01*g/mole);

//Create Aluminum57S (Reactor Calandria)
Aluminum57S = new G4Material("Aluminuum 57S", 2.7*g/cm3, 8, kStateSolid);
Aluminum57S->AddElement(Al, 96.7*perCent);
Aluminum57S->AddElement(Si, 0.25*perCent);
Aluminum57S->AddElement(Fe, 0.4*perCent);
Aluminum57S->AddElement(Cu, 0.1*perCent);
Aluminum57S->AddElement(Mn, 0.1*perCent);
Aluminum57S->AddElement(Mg, 2.2*perCent);
Aluminum57S->AddElement(Cr, 0.15*perCent);
Aluminum57S->AddElement(Zn, 0.1*perCent);

//Create AlPresT (pressure Tube)
AlPresT = new G4Material("Aluminuum 6061", 2.712631*g/cm3, 8, kStateSolid);

AlPresT->AddElement(Al, 99.1244424*perCent);
AlPresT->AddElement(Si, 0.5922414*perCent);
AlPresT->AddElement(Fe, 0.1211379*perCent);
AlPresT->AddElement(Cu, 0.0018171*perCent);
```

```
AlPresT->AddElement(Mn, 0.0383626*perCent);
AlPresT->AddElement(Cr, 0.1211405*perCent);
AlPresT->AddElement(Li, 0.00075712*perCent);
AlPresT->AddElement(B, 0.00010095*perCent);


//Create AlCalT (calandria Tube)
AlCalT = new G4Material("Aluminuum 6063", 2.684951*g/cm3, 8, kStateSolid);
AlCalT->AddElement(Al, 99.18675267*perCent);
AlCalT->AddElement(Si, 0.509640251*perCent);
AlCalT->AddElement(Fe, 0.241396625*perCent);
AlCalT->AddElement(Li, 0.00754387*perCent);
AlCalT->AddElement(B, 0.000100586*perCent);
AlCalT->AddElement(Mn, 0.041228175*perCent);
AlCalT->AddElement(Gd, 0.000010059*perCent);
AlCalT->AddElement(Ti, 0.041228175*perCent);


Moderator = new G4Material("Moderator", 1.102597*g/cm3, 2, kStateLiquid, 299.51*kelvin);
Moderator->AddMaterial(D2O, 98.705*perCent);
Moderator->AddMaterial(H2O, 1.295*perCent);


//Create Annulus Gas
AnnulusGas = new G4Material("AnnulusGas", 0.0012*g/cm3, 2, kStateGas/*, 448.72*kelvin*/);
AnnulusGas->AddElement(C,27.11*perCent);
AnnulusGas->AddElement(Oxygen,72.89*perCent);


Zr4 = new G4Material("Zircaloy-4", 6.55*g/cm3, 4, kStateSolid);
Zr4->AddElement(Oxygen, 0.12*perCent);
```

```
Zr4->AddElement(CrZr, 0.11*perCent);
Zr4->AddElement(FeZr, 0.22*perCent);
Zr4->AddElement(Zr, 99.58*perCent);


// Make Air
Air = new G4Material("Air", 1.29*mg/cm3, 5, kStateGas);
Air->AddElement(N, 74.74095914*perCent);
Air->AddElement(Oxygen, 23.49454694*perCent);
Air->AddElement(Ar, 1.274547311*perCent); Air->AddElement(Li, 0.474350981*per-
Cent);
Air->AddElement(C, 0.015595629*perCent);


// Add materials
matMap["World"] = World;
matMap["LEUMat"] = LEUMat;
matMap["Graphite"] = Graphite;
matMap["Al57S"] = Aluminum57S;
matMap["AlPresT"] = AlPresT;
matMap["AlCalT"] = AlCalT;
matMap["Zr4"] = Zr4;
matMap["Air"] = Air;
matMap["Moderator"] = Moderator;
matMap["Coolant"] = H2O;
matChanged = false;
return;
}
```

Appendix C

Log File

——————————————————————————

Geant 4 Simulation of Neutron Stability

#

Geant4 version Name: geant4-09-06-patch-02 (17-May-2013)

Neutron Stability rev.0.9.3 (Bazaar build date 2013-03-13 08:01:38 -0400)

#

Current time: Tue Feb 17 13:13:02 2015

#

World Choice:

World: Q_ZED2

Reactor Material: 0

Shannon entropy mesh: (20, 20, 20)

#

Run Options:

Number of runs: 200

Number of primaries per Event: 11250

Number of events: 80

Run duration (ns): 1e+06

Initial Neutron Energy (MeV): 2

#

World Properties:

#

Nuclear Data Options:

Data Library:

/home/mahzoos/geant4.9.6-install/share/Geant4-9.6.2/data/G4NDL4.2

Cross section temperature (K): 0

#

Log Files:

Logging File: Log_Sub114_Q_ZED2.txt

Output source file: Src_Sub114_Q_ZED2.txt

Save source distribution interval: 50

Output fission data file: Fiss_Sub114_Q_ZED2.txt

#

Initialization time: User=82.06s Real=170.64s Sys=17.9s

#

————————————————————————————

#

Starting Simulation

#

Number of Primaries per Run = 900000

Number of Events per Run = 80

#

#

Interpolation started at run 50

Run	Start (us)	Lifetime (us)	Production	Loss	krun	keff	FS Shannon H	S Shannon H
74	3.65E+04	4384	65603	65812	0.9979	0.9968	67.2818	95.98
75	3.70E+04	4705	66273	66751	0.9952	0.9928	67.2484	95.9901
76	3.75E+04	3968	66931	66982	0.9995	0.9992	67.2733	95.946
77	3.80E+04	3961	66861	66721	1.0014	1.0021	67.1755	95.9392
78	3.85E+04	1.17E+04	66870	66783	1.0009	1.0013	67.2828	95.9386
79	3.90E+04	2188	67481	67268	1.0021	1.0032	67.2167	95.9249
80	3.95E+04	7280	67410	66979	1.0043	1.0064	67.316	95.9341
81	4.00E+04	2512	65659	66408	0.9925	0.9887	67.2438	95.9619
82	4.05E+04	1917	66201	66526	0.9968	0.9951	67.3232	95.938
83	4.10E+04	6584	65857	66318	0.9954	0.993	67.2709	95.9836
84	4.15E+04	5222	64489	65481	0.9901	0.9849	67.2245	96.0378
85	4.20E+04	3769	66302	66516	0.9979	0.9968	67.2728	96.0246
86	4.25E+04	7992	66512	66295	1.0022	1.0033	67.2296	95.9617
87	4.30E+04	3798	67916	67407	1.0051	1.0076	67.2496	95.9378
88	4.35E+04	6873	65940	66658	0.9928	0.9892	67.145	95.9479
89	4.40E+04	6899	66257	66334	0.9992	0.9988	67.2981	95.9489
90	4.45E+04	5327	66421	66501	0.9992	0.9988	67.23	95.9416
91	4.50E+04	5935	65312	65690	0.9962	0.9942	67.2022	95.963
92	4.55E+04	4377	66459	66785	0.9967	0.9951	67.2677	95.9755
93	4.60E+04	3197	66943	66815	1.0013	1.0019	67.2912	95.8984
94	4.65E+04	4997	66129	66341	0.9979	0.9968	67.1732	95.9398
95	4.70E+04	1957	66345	66574	0.9977	0.9966	67.3072	95.9263
96	4.75E+04	4715	66139	66231	0.9991	0.9986	67.1486	95.9869
97	4.80E+04	2716	67229	66872	1.0036	1.0053	67.2047	95.9562
98	4.85E+04	3005	65051	65942	0.9911	0.9865	67.2178	95.9778
99	4.90E+04	4169	66151	66360	0.9979	0.9969	67.317	95.9916
100	4.95E+04	4643	65878	66493	0.9939	0.9908	67.2792	95.9748
# Avg (last 37 runs):	40.36	537490	597459	0.933351	0.899621	58.9123	88.9221	2.3e+03

Source convergence limit = 2%

Source converged after 50 runs.

#

Total computation time: User=3.9e+03s Real=4.6e+05s Sys=2.5e+02s

Appendix D

Source File

Uvec 1878463799 149 1687352324 124790585 # Source file for following input:

#

————————————————————————————

#

Geant 4 Simulation of Neutron Stability

#

Geant4 version Name: geant4-09-06-patch-02 (17-May-2013)

Neutron Stability rev.0.9.3 (Bazaar build date 2013-03-13 08:01:38 -0400)

#

Current time: Fri Jan 23 14:45:47 2015

#

World Choice:

World: Q_ZED2

Reactor Material: 0

Shannon entropy mesh: (20, 20, 20)

#

Run Options:

Number of runs: 100

Number of primaries per Event: 1000

Number of events: 4

Run duration (ns): 100000

Initial Neutron Energy (MeV): 2

#

World Properties:

#

Nuclear Data Options:

Data Library:

/home/salmamah/geant4.9.6-install/share/Geant4-9.6.2/data/G4NDL4.2

Cross section temperature (K): 0

#

Log Files:

Logging File: Log_ZED2_qmoren.txt

Output source file: Src_ZED2_GraphiteTest.txt

Save source distribution interval: 1

Output fission data file: Fiss_ZED2_qmorenlongertod.txt

#

#

Source distribution after 9 runs

900000

3916

900000 6.8653236593135749e+05

-9.9647743721373763e+02 -1.0244133034371525e+03 -1.7714993419830464e+03

-3.6778566063639857e-03 -4.3646228882421928e-03 7.6166935298490758e-05

7.9286222949231000e-01 8.7348099015113834e-01 1.9369974052051980e-01

1.9379515821094258e+00 1.0000000000000000e+00

Above values are defined below in order they are saved in the file

Global Time [ns]	The current simulation time the other data was recorded at that time.
Lifetime [ns]	The total simulation time since the hit that created the neutron. For delayed neutrons, this is the initial fission that set off the decay chain.
Psition (x,y,z) [mm]	The position of the neutron at the global time.
Momentum(x,y,z) [Mev/c]	The momentum of the neutron at the global time.
η (x,y,z) [mm]	theNumberOfInteractionLengthLeft for the hadronic processes.
Discretionary	Energy/weight/etc

Bibliography

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

[2] W. M. Stacey, *Nuclear Reactor Physics*. John Wiley & Sons, 2001.

[3] E.E.Lewis and J. W.F. Miller, *Computational methods of neutron transport*. New York: John Wiley & Sons, 1984.

[4] J. Atfield, S. Yue, and M. Zeller, "Subcritical reactivity measurements in ZED-2," 2013.

[5] J. Atfield, "28-element natural uo2 fuel assemblies in ZED-2," tech. rep., ZED-2 HWR EXP 001, in OECD-NEA Nuclear Safety Committee at AECL, 2011. International Handbook of Evaluated Reactor Physics Benchmark Experiments.

[6] L. Russell, "Simulation of time-dependent neutron populations for reactor physics applications using the geant4 monte carlo toolkit," Master's thesis, McMaster University, Department of Engineering Physics, 2012.

[7] L. Russell, A. Buijs, and G. Jonkmans, "G4-STORK: a Monte Carlo reactor kinetics simulation code," *Nuclear Science and Engineering*, 2013.

[8] J. Leppänen, *Development of a New Monte Carlo Reactor Physics Code*. PhD thesis, Helsinki University of Technology, 2007.

[9] P. Rinard, "Neutron interaction with matter," *in Passive Nondestructive Assay of Nuclear Materials*, p. 357, 1991.

[10] J. Leppänen *et al.*, *Development of a new Monte Carlo reactor physics code*. VTT Technical Research Centre of Finland, 2007.

[11] R. P. Dermott E. Cullen, Christopher J. Clouse and R. C. Little, "Static and dynamic criticality: Are they different?," November 2003.

[12] J. C. Wagner, *Monte Carlo transport calculations and analysis for reactor pressure vessel neutron fluence*. PhD thesis, The Pennsylvania State University, 1994.

[13] R. Ben, "Solving the diffusion equation numerically." University Lecture, 2014.

[14] J. B. Taylor, *The development of a three-dimensional nuclear reactor kinetics methodology based on the method of characteristics*. ProQuest, 2007.

[15] K. O. Ott and R. J. Neuhold, *Introductory nuclear reactor dynamics*. American Nuclear Society La Grange Park, 1985.

[16] L. L. Carter and E. D. Cashwell, "Particle-transport simulation with the monte carlo method," tech. rep., Los Alamos Scientific Lab., N. Mex.(USA), 1975.

[17] J. F. Briesmeister *et al.*, "MCNP-A general Monte Carlo N-particle transport code," *Version 4C, LA-13709-M, Los Alamos National Laboratory*, 2000.

[18] R. Forster and T. Godfrey, "MCNP-a general Monte Carlo code for neutron and photon transport," in *Monte-Carlo Methods and Applications in Neutronics, Photonics and Statistical Physics*, pp. 33–55, Springer, 1985.

[19] S. Agostinelli, J. Allison, K. a. Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, D. Axen, S. Banerjee, G. Barrand, *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.

[20] J. Allison, R. Brun, F. Bruyant, F. Bullock, C. Chang, J. Dumont, P. Hattersley, R. Hemingway, P. Hobson, D. Hochman, *et al.*, "An application of the GEANT3 geometry package to the description of the opal detector," *Computer Physics Communications*, vol. 47, no. 1, pp. 55–74, 1987.

[21] N. P. A. J. G.Cosmo, S.Giani *et al.*, "GEANT4: An object-oriented toolkit for simulation in HEP," *CERN/LHCC*, pp. 98–44, 1998.

[22] G. Collaboration, "Introduction to geant4," 2012.

[23] W. Naing, M. Tsuji, and Y. Shimazu, "Subcriticality measurement of pressurized water reactors by the modified neutron source multiplication method," *Journal of Nuclear Science and Technology*, vol. 40, no. 12.

[24] D. E. Cullen, "TART2012 an overview of a coupled neutron-photon 3-D, combinatorial geometry time dependent Monte Carlo transport code," 2012.

[25] J. Leppänen, "Serpent–a Continuous-energy Monte Carlo Reactor Physics Burnup Calculation Code," 2012.

[26] *Code distribution and data libraries*. Available at http://montecarlo.vtt.fi/users.htm.

[27] G. Cooperman, V. H. Nguyen, and I. Malioutov, "Parallelization of geant4 using top-c and marshalgen," in *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*, pp. 48–55, IEEE, 2006.

[28] J.P.Wellisch, "THE NETRON_HP NETRON TRANSPORT CODE.," *American Nuclear Society La Grange Park*, 2005.

[29] E. Mendoza, D. Cano-Ott, C. Guerrero, and R. Capote, "New evaluated neutron cross section libraries for the geant4 code," tech. rep., International Atomic Energy Agency, International Nuclear Data Committee, Vienna (Austria), 2012.

[30] G. Yesilyurt, *Advanced Monte Carlo methods for analysis of very high temperature reactors: On-the-fly Doppler broadening and deterministic/Monte Carlo methods*. PhD thesis, Los Alamos National Laboratory, 2009.