STOCHASTIC POINT KINETICS EQUATIONS IN
NUCLEAR REACTOR DYNAMICS

by

JAMES G HAYES, B.S.

A THESIS

IN

MATHEMATICS

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

Approved

Edward J. Allen
Chairperson of the Committee

Harold Dean Victory

Padmanabhan Seshaiyer

Accepted

John Borrelli
Dean of the Graduate School

May, 2005

ACKNOWLEDGEMENTS

I would like to thank the entire mathematics department, especially Dr. Edward Allen, at Texas Tech University through this entire process. Their continuing support and dedication remains second to none. To my fellow graduate students, without you, graduate school would be an almost insurmountable task. Also, I would like to thank the mathematics department at Southwest Missouri State University for encouraging me to continue my higher education. Last but certainly not least, to my family, you have shown me nothing but love and support. It means more to me than you probably know.

CONTENTS

ACKNOWLEDGEMENTS	ii
ABSTRACT	iv
LIST OF TABLES	v
LIST OF FIGURES	vi
I INTRODUCTION	1
II MODEL FORMULATION	3
III NUMERICAL APPROXIMATION	11
IV COMPUTATIONAL RESULTS	15
V SUMMARY AND CONCLUSIONS	21
BIBLIOGRAPHY	22
APPENDIX	24
A STOPCA.M	24
B STOHIST.M	40

ABSTRACT

A system of Itô stochastic differential equations is derived that model the dynamics of the neutron density and the delayed neutron precursors in a point nuclear reactor. The stochastic model is tested against Monte Carlo calculations and experimental data. The results demonstrate that the stochastic differential equation model accurately describes the random behavior of the neutron density and the precursor concentrations in a point reactor.

LIST OF TABLES

4.1	A comparison using only one precursor	17
4.2	A comparison using a given neutron level, $n_{level} = 4000$	17
4.3	A comparison using a prompt subcritical step reactivity, $\rho = 0.003$. .	17
4.4	A comparison using prompt critical step reactivity, $\rho = 0.007$	18
4.5	A comparison for a Godiva experiment	18

LIST OF FIGURES

4.1	Neutron density using a prompt subcritical step reactivity, $\rho = 0.003$	18
4.2	Precursor density using a prompt subcritical step reactivity, $\rho = 0.003$	19
4.3	Histogram of times for the Godiva reactor to reach 4200 neutrons . .	19
4.4	Frequency histograms for the experimental measurements and calculated results for the Godiva reactor	20

CHAPTER I

INTRODUCTION

The point-kinetics equations [1, 2, 3, 9, 12, 15] model the time-dependent behavior of a nuclear reactor. Computational solutions of the point-kinetics equations provide insight into the dynamics of nuclear reactor operation and are useful, for example, in understanding the power fluctuations experienced during start-up or shut-down when the control rods are adjusted. The point-kinetics equations are a system of differential equations for the neutron density and for the delayed neutron precursor concentrations. (Delayed neutron precursors are specific radioactive isotopes which are formed in the fission process and decay through neutron emission.) The neutron density and delayed neutron precursor concentrations determine the time-dependent behavior of the power level of a nuclear reactor and are influenced, for example, by control rod position.

The point-kinetics equations are deterministic and can only be used to estimate average values of the neutron density, the delayed neutron precursor concentrations, and power level. However, the actual dynamical process is stochastic in nature and the neutron density and delayed neutron precursor concentrations vary randomly with time. At high power levels, the random behavior is negligible but at low power levels, such as at start-up [10, 11, 14], random fluctuations in the neutron density and neutron precursor concentrations can be significant.

The point-kinetics equations actually model a system of interacting populations, specifically, the populations of neutrons and delayed neutron precursors. After identifying the physical dynamical system as a population process, techniques developed in [4, 5] can be applied to transform the deterministic point-kinetics equations into a stochastic differential equation system that accurately models the random behavior of the process. The stochastic system of differential equations generalize the deterministic point-kinetics equations.

Computational solution of these stochastic equations is performed in the present investigation by applying a modified form of the numerical method developed in [12]. The stochastic model is tested against Monte Carlo calculations and experimental data. The computational results indicate that the stochastic model and computational method are accurate.

CHAPTER II

MODEL FORMULATION

It is first useful to derive the point-kinetic equations in order to separate the birth and death processes of the neutron populations. This will help us in formulating the stochastic model. Following the derivation presented in [9], the deterministic time-dependent equations satisfied by the neutron density and the delayed neutron precursors can be described by

$$\frac{\partial N}{\partial t} = Dv\nabla^2 N - (\Sigma_a - \Sigma_f)vN + [(1 - \beta)k_\infty\Sigma_a - \Sigma_f]vN + \sum_i \lambda_i C_i + S_0 \quad (2.1)$$

$$\frac{\partial C_i}{\partial t} = \beta_i k_\infty \Sigma_a v N - \lambda_i C_i \quad (2.2)$$

for $i = 1, 2, \dots, m$ where $N(r, t)$ is the density of neutrons, r is position, t is time, v is the velocity, and $Dv\nabla^2 N$ is a term accounting for diffusion of the neutrons. The absorption and fission cross sections are Σ_a and Σ_f , respectively. The capture cross section is $\Sigma_a - \Sigma_f$. The prompt-neutron contribution to the source is $[(1 - \beta)k_\infty\Sigma_a - \Sigma_f]vN$ where $\beta = \sum_{i=1}^m \beta_i$ is the delayed-neutron fraction and $(1 - \beta)$ is the prompt-neutron fraction. The infinite medium reproduction factor is k_∞ . The rate of transformations from the neutron precursors to the neutron population is $\sum_{i=1}^m \lambda_i C_i$ where λ_i is the delay constant and $C_i(r, t)$ is the density of the i^{th} type of precursor, for $i = 1, 2, 3, \dots, m$. Finally, extraneous neutron sources are represented by $S_0(r, t)$.

In the present investigation, neutron captures are considered deaths. The fission process here is considered as a “pure” birth process where $\nu(1 - \beta) - 1$ neutrons are born in each fission along with the precursor fraction $\nu\beta$. For a single energy group model, a neutron is lost in each fission, but $\nu(1 - \beta)$ neutrons are immediately gained with the overall result that $\nu(1 - \beta) - 1$ neutrons are immediately “born” in the energy group. However, in a multiple group model, a fission event would be treated as a death of a neutron in the energy group of the neutron causing the fission along with the simultaneous birth of $\nu(1 - \beta)$ neutrons in several high energy groups.

As in [9], let $N = f(r)n(t)$ and $C_i = g_i(r)c_i(t)$ where we assume that N and C_i are separable in time and space. Now, (2.2) becomes

$$\frac{dc_i}{dt} = \beta_i k_\infty \Sigma_a v \frac{f(r)n(t)}{g_i(r)} - \lambda_i c_i(t).$$

Note that it is assumed $\frac{f}{g_i}$ is independent of time. We also assume that $\frac{f(r)}{g_i(r)} = 1$. Thus we have

$$\frac{dc_i}{dt} = \beta_i k_\infty \Sigma_a v n - \lambda_i c_i(t). \quad (2.3)$$

By making the same substitutions as above, (2.1) becomes

$$\frac{dn}{dt} = Dv \frac{\nabla^2 f}{f} n(t) - (\Sigma_a - \Sigma_f) v n(t) + [(1 - \beta) k_\infty \Sigma_a - \Sigma_f] v n(t) + \sum_i \lambda_i \frac{g_i c_i}{f} + \frac{S_0}{f}.$$

We assume that f satisfies $\nabla^2 f + B^2 f = 0$ (a Helmholtz equation) and that S_0 has the same spatial dependence as f . Thus $q(t) = \frac{S_0(r,t)}{f(r)}$. The above equation describing the rate of change of neutrons with time is

$$\frac{dn}{dt} = -Dv B^2 n - (\Sigma_a - \Sigma_f) v n + [(1 - \beta) k_\infty \Sigma_a - \Sigma_f] v n + \sum_i \lambda_i c_i + q. \quad (2.4)$$

We now consider these equations as representing a population process where $n(t)$ is the population of neutrons and $c_i(t)$ is the population of the i^{th} precursor. We separate the neutron reactions into two terms, deaths and births. Therefore,

$$\frac{dn}{dt} = \underbrace{-Dv B^2 n - (\Sigma_a - \Sigma_f) v n}_{\text{deaths}} + \underbrace{(k_\infty \Sigma_a - \Sigma_f) v n}_{\text{births}} - \underbrace{\beta k_\infty \Sigma_a v n + \sum_i \lambda_i c_i}_{\text{transformations}} + q \quad (2.5)$$

$$\frac{dc_i}{dt} = \beta_i k_\infty \Sigma_a v n - \lambda_i c_i. \quad (2.6)$$

The following symbols are introduced to help simplify the above system. The absorption lifetime and the diffusion length are represented as $\ell_\infty = \frac{1}{v \Sigma_a}$ and $L^2 = \frac{D}{\Sigma_a}$. Equation (2.5) becomes

$$\frac{dn}{dt} = \frac{-L^2 B^2}{\ell_\infty} n - \frac{(\Sigma_a - \Sigma_f)}{\Sigma_a \ell_\infty} n + \frac{(k_\infty \Sigma_a - \Sigma_f)}{\Sigma_a \ell_\infty} n - \frac{\beta k_\infty}{\ell_\infty} n + \sum_i \lambda_i c_i + q.$$

After simplification and regrouping, the above equation becomes

$$\frac{dn}{dt} = \underbrace{\left[\frac{-L^2 B^2 - \frac{(\Sigma_a - \Sigma_f)}{\Sigma_a}}{\ell_\infty} \right]}_{\text{deaths}} n + \underbrace{\left[\frac{k_\infty - \frac{\Sigma_f}{\Sigma_a}}{\ell_\infty} \right]}_{\text{births}} n - \underbrace{\frac{\beta k_\infty}{\ell_\infty} n + \sum_i \lambda_i c_i + q}_{\text{transformations}}. \quad (2.7)$$

Performing the same substitutions in (2.6) gives

$$\frac{dc_i}{dt} = \frac{\beta_i k_\infty}{\ell_\infty} n - \lambda_i c_i. \quad (2.8)$$

Two more constants $k = \frac{k_\infty}{1+L^2 B^2}$ and $\ell_0 = \frac{\ell_\infty}{1+L^2 B^2}$ are now introduced which are the reproduction factor and neutron lifetime. Equation (2.7) then becomes

$$\frac{dn}{dt} = \left[\frac{(-1 - L^2 B^2) + \frac{\Sigma_f}{\Sigma_a}}{\ell_\infty} \right] n + \left[\frac{k_\infty}{\ell_\infty} - \frac{\Sigma_f}{\Sigma_a \ell_\infty} \right] n - \frac{\beta k_\infty}{\ell_\infty} n + \sum_i \lambda_i c_i + q.$$

After substitution and rearranging the above equation, we have

$$\frac{dn}{dt} = \left[\frac{-1}{\ell_0} + \frac{\Sigma_f}{\Sigma_a \ell_\infty} \right] n + \left[\frac{k}{\ell_0} - \frac{\Sigma_f}{\Sigma_a \ell_\infty} \right] n - \frac{\beta k}{\ell_0} n + \sum_i \lambda_i c_i + q. \quad (2.9)$$

We also make the above substitutions to (2.8) to obtain

$$\frac{dc_i}{dt} = \frac{\beta_i k}{\ell_0} n - \lambda_i c_i. \quad (2.10)$$

Next, we consider these equations in terms of neutron generation time [9]. We define $\ell = \frac{\ell_0}{k}$ as generation time. Now (2.9) and (2.10) become

$$\frac{dn}{dt} = \left[\frac{\frac{-1}{k}}{\ell} + \frac{\Sigma_f}{\Sigma_a \ell_\infty} \right] n + \left[\frac{1}{\ell} - \frac{\Sigma_f}{\Sigma_a \ell_\infty} \right] n - \frac{\beta}{\ell} n + \sum_i \lambda_i c_i + q \quad (2.11)$$

$$\frac{dc_i}{dt} = \frac{\beta_i}{\ell} n - \lambda_i c_i. \quad (2.12)$$

We now introduce reactivity which is defined as $\rho = 1 - \frac{1}{k}$. The first equation becomes

$$\frac{dn}{dt} = \left[\frac{\rho - 1}{\ell} + \frac{\Sigma_f}{\Sigma_a \ell_\infty} \right] n + \left[\frac{1}{\ell} - \frac{\Sigma_f}{\Sigma_a \ell_\infty} \right] n - \frac{\beta}{\ell} n + \sum_i \lambda_i c_i + q \quad (2.13)$$

For further simplification, we consider the term $\frac{\Sigma_f}{\Sigma_a \ell_\infty}$.

$$\frac{\Sigma_f}{\Sigma_a \ell_\infty} = \frac{\Sigma_f}{\Sigma_a \ell_0 (1 + L^2 B^2)} = \frac{\Sigma_f}{\Sigma_a \ell_0 \frac{k_\infty}{k}} = \frac{\Sigma_f}{\Sigma_a \ell k_\infty} = \frac{1}{\ell} \frac{\Sigma_f}{\Sigma_a k_\infty} = \frac{\alpha}{\ell}$$

where α is defined as $\alpha = \frac{\Sigma_f}{\Sigma_a k_\infty}$. Equation (2.13) becomes

$$\frac{dn}{dt} = \left[\frac{\rho - 1 + \alpha}{\ell} \right] n + \left[\frac{1 - \alpha - \beta}{\ell} \right] n + \sum_i \lambda_i c_i + q. \quad (2.14)$$

The final deterministic system becomes

$$\frac{dn}{dt} = - \left[\frac{-\rho + 1 - \alpha}{\ell} \right] n + \left[\frac{1 - \alpha - \beta}{\ell} \right] n + \sum_{i=1}^m \lambda_i c_i + q \quad (2.15)$$

$$\frac{dc_i}{dt} = \frac{\beta_i}{\ell} n - \lambda_i c_i \quad (2.16)$$

for $i = 1, 2, \dots, m$ where the birth and death processes are separated in (2.15).

Note that n is the population size of neutrons, and c_i is the population size of the i^{th} neutron precursor. The neutron birth rate due to fission is $b = \frac{1 - \alpha - \beta}{\ell(-1 + (1 - \beta)\nu)}$, where the denominator has the term $(-1 + (1 - \beta)\nu)$ which is the number of new neutrons “born” in each fission. The neutron death rate due to captures and leakage is $d = \frac{-\rho + 1 - \alpha}{\ell}$. Also, $\lambda_i c_i$ is the rate that the i^{th} precursor is transformed into neutrons, and q is the rate that source neutrons are produced.

In order to derive the stochastic system for the population dynamics, we consider first for simplicity just one precursor. Notice for one precursor that $\beta = \beta_1$, where β is used for one precursor to represent the total delayed neutron fraction. This notation makes it easier to extend the model to several precursors. The stochastic system will be later generalized to m precursors. We have for one precursor:

$$\begin{aligned} \frac{dn}{dt} &= - \left[\frac{-\rho + 1 - \alpha}{\ell} \right] n + \left[\frac{1 - \alpha - \beta}{\ell} \right] n + \lambda_1 c_1 + q \\ \frac{dc_1}{dt} &= \frac{\beta_1}{\ell} n - \lambda_1 c_1. \end{aligned}$$

Now consider a very small time interval Δt where the probability of more than one event occurring during time Δt is small. During time Δt , there are four different possibilities for an event. Let $[\Delta n, \Delta c_1]^T$ be the change in the populations n and c_1 in time Δt . We assume in the present investigation that the changes are approximately

normally distributed. The four possibilities for $[\Delta n, \Delta c_1]^T$ are:

$$\begin{aligned} \begin{bmatrix} \Delta n \\ \Delta c_1 \end{bmatrix}_1 &= \begin{bmatrix} -1 \\ 0 \end{bmatrix} \\ \begin{bmatrix} \Delta n \\ \Delta c_1 \end{bmatrix}_2 &= \begin{bmatrix} -1 + (1 - \beta)\nu \\ \beta_1\nu \end{bmatrix} \\ \begin{bmatrix} \Delta n \\ \Delta c_1 \end{bmatrix}_3 &= \begin{bmatrix} 1 \\ -1 \end{bmatrix} \\ \begin{bmatrix} \Delta n \\ \Delta c_1 \end{bmatrix}_4 &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} \end{aligned}$$

where the first event represents a death (capture), the second event represents a fission event with $-1 + (1 - \beta)\nu$ neutrons produced and $\beta_1\nu$ delayed neutrons precursors produced, the third event represents a transformation of a delayed neutron precursor to a neutron, and the fourth event represents a birth of a source neutron.

The probabilities of these events are:

$$\begin{aligned} P_1 &= dn\Delta t \\ P_2 &= bn\Delta t = \frac{1}{\nu\ell}n\Delta t \\ P_3 &= \lambda_1 c_1 \Delta t \\ P_4 &= q\Delta t \end{aligned}$$

where it is assumed that the extraneous source produces neutrons randomly following a Poisson process with intensity q .

We now consider the mean change and the covariance of the change for a small time interval Δt . We have to $O((\Delta t)^2)$:

$$E \left(\begin{bmatrix} \Delta n \\ \Delta c_1 \end{bmatrix} \right) = \sum_{k=1}^4 P_k \begin{bmatrix} \Delta n \\ \Delta c_1 \end{bmatrix}_k = \begin{bmatrix} \frac{\rho-\beta}{\ell}n + \lambda_1 c_1 + q \\ \frac{\beta_1}{\ell}n - \lambda_1 c_1 \end{bmatrix} \Delta t.$$

Note that

$$(\beta_1\nu)P_2 = \frac{\beta_1}{\ell}n\Delta t$$

and thus

$$b = \frac{1 - \frac{1}{\nu} - \beta}{\ell(-1 + (1 - \beta)\nu)} = \frac{\nu - 1 - \beta\nu}{\nu\ell(-1 + (1 - \beta)\nu)} = \frac{1}{\nu\ell}$$

assuming that $\alpha = \frac{1}{\nu}$. Therefore,

$$E \left(\begin{bmatrix} \Delta n \\ \Delta c_1 \end{bmatrix} \begin{bmatrix} \Delta n & \Delta c_1 \end{bmatrix} \right) = \sum_{k=1}^4 P_k \begin{bmatrix} \Delta n \\ \Delta c_1 \end{bmatrix}_k \begin{bmatrix} \Delta n & \Delta c_1 \end{bmatrix}_k = \widehat{B} \Delta t$$

where

$$\widehat{B} = \begin{bmatrix} \gamma n + \lambda_1 c_1 + q & \frac{\beta_1}{\ell}(-1 + (1 - \beta)\nu)n - \lambda_1 c_1 \\ \frac{\beta_1}{\ell}(-1 + (1 - \beta)\nu)n - \lambda_1 c_1 & \frac{\beta_1^2 \nu}{\ell}n + \lambda_1 c_1 \end{bmatrix},$$

and

$$\gamma = \frac{-1 - \rho + 2\beta + (1 - \beta)^2 \nu}{\ell}.$$

Recalling the assumption that the changes are approximately normally distributed, the above result implies that to $O((\Delta t)^2)$

$$\begin{bmatrix} n(t + \Delta t) \\ c_1(t + \Delta t) \end{bmatrix} = \begin{bmatrix} n(t) \\ c_1(t) \end{bmatrix} + \begin{bmatrix} \frac{\rho - \beta}{\ell}n + \lambda_1 c_1 \\ \frac{\beta_1}{\ell}n - \lambda_1 c_1 \end{bmatrix} \Delta t + \begin{bmatrix} q \\ 0 \end{bmatrix} \Delta t + \widehat{B}^{\frac{1}{2}} \sqrt{\Delta t} \begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix},$$

where $\eta_1, \eta_2 \sim N(0, 1)$ and $\widehat{B}^{\frac{1}{2}}$ is the square root of the matrix $\widehat{B}$, i.e. $\widehat{B} = \widehat{B}^{\frac{1}{2}} \cdot \widehat{B}^{\frac{1}{2}}$.

This equation gives, as $\Delta t \rightarrow 0$, the following Itô stochastic system

$$\frac{d}{dt} \begin{bmatrix} n \\ c_1 \end{bmatrix} = \widehat{A} \begin{bmatrix} n \\ c_1 \end{bmatrix} + \begin{bmatrix} q \\ 0 \end{bmatrix} + \widehat{B}^{\frac{1}{2}} \frac{d\vec{W}}{dt} \quad (2.17)$$

where

$$\widehat{A} = \begin{bmatrix} \frac{\rho - \beta}{\ell} & \lambda_1 \\ \frac{\beta_1}{\ell} & -\lambda_1 \end{bmatrix},$$

$$\widehat{B} = \begin{bmatrix} \gamma n + \lambda_1 c_1 + q & \frac{\beta_1}{\ell}(-1 + (1 - \beta)\nu)n - \lambda_1 c_1 \\ \frac{\beta_1}{\ell}(-1 + (1 - \beta)\nu)n - \lambda_1 c_1 & \frac{\beta_1^2 \nu}{\ell}n + \lambda_1 c_1 \end{bmatrix},$$

and

$$\vec{W}(t) = \begin{bmatrix} W_1(t) \\ W_2(t) \end{bmatrix}$$

where $W_1(t)$ and $W_2(t)$ are Wiener processes.

Now consider m precursors. Let

$$\widehat{A} = \begin{bmatrix} \frac{\rho-\beta}{\ell} & \lambda_1 & \lambda_2 & \cdots & \lambda_m \\ \frac{\beta_1}{\ell} & -\lambda_1 & 0 & \cdots & 0 \\ \frac{\beta_2}{\ell} & 0 & -\lambda_2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ \frac{\beta_m}{\ell} & 0 & \cdots & 0 & -\lambda_m \end{bmatrix}$$

and

$$\widehat{B} = \begin{bmatrix} \zeta & a_1 & a_2 & \cdots & a_m \\ a_1 & r_1 & b_{2,3} & \cdots & b_{2,m} \\ a_2 & b_{3,2} & r_2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & b_{m-1,m} \\ a_m & b_{m,2} & \cdots & b_{m,m-1} & r_m \end{bmatrix} \quad (2.18)$$

where

$$\begin{aligned} \zeta &= \gamma n + \sum_{i=1}^m \lambda_i c_i + q, \\ \gamma &= \frac{-1 - \rho + 2\beta + (1 - \beta)^2 \nu}{\ell}, \\ a_i &= \frac{\beta_i}{\ell} (-1 + (1 - \beta) \nu) n - \lambda_i c_i, \\ b_{i,j} &= \frac{\beta_{i-1} \beta_{j-1} \nu}{\ell} n, \end{aligned}$$

and

$$r_i = \frac{\beta_i^2 \nu}{\ell} n + \lambda_i c_i.$$

The same approach, but for m precursors, yields the Itô stochastic system:

$$\frac{d}{dt} \begin{bmatrix} n \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} = \widehat{A} \begin{bmatrix} n \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} + \begin{bmatrix} q \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \widehat{B}^{\frac{1}{2}} \frac{d\vec{W}}{dt}. \quad (2.19)$$

Equation (2.19) is the final stochastic point-kinetics equations. Note that if $\widehat{B} = 0$, then (2.19) reduces to the standard deterministic point- kinetics equations; therefore (2.19) can be considered a generalization of the standard point-kinetics model.

CHAPTER III

NUMERICAL APPROXIMATION

Notice that the stochastic point-kinetics equations for m delayed groups have the form:

$$\frac{d\vec{x}}{dt} = A\vec{x} + B(t)\vec{x} + \vec{F}(t) + \widehat{B}^{\frac{1}{2}} \frac{d\vec{W}}{dt} \quad (3.1)$$

where $\widehat{B}$ is given in (2.18),

$$\vec{x} = \begin{bmatrix} n \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}, \quad (3.2)$$

A is the $(m+1) \times (m+1)$ matrix:

$$A = \begin{bmatrix} \frac{-\beta}{\ell} & \lambda_1 & \lambda_2 & \cdots & \lambda_m \\ \frac{\beta_1}{\ell} & -\lambda_1 & 0 & \cdots & 0 \\ \frac{\beta_2}{\ell} & 0 & -\lambda_2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ \frac{\beta_m}{\ell} & 0 & \cdots & 0 & -\lambda_m \end{bmatrix}, \quad (3.3)$$

B is the $(m+1) \times (m+1)$ matrix:

$$B(t) = \begin{bmatrix} \frac{\rho(t)}{\ell} & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}, \quad (3.4)$$

and, $F(t)$ is given as:

$$\vec{F}(t) = \begin{bmatrix} q(t) \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \quad (3.5)$$

Note that matrix $\hat{A}$ from (2.19) has been separated into matrices A and $B(t)$ in (3.1), that is, $\hat{A} = A + B(t)$ where A is a constant matrix.

Generally, the source and reactivity functions vary slowly with respect to time with relation to the neutron density. Therefore, we will approximate these functions using a piecewise constant approximation. We let

$$\rho(t) \approx \rho \left(\frac{t_i + t_{i+1}}{2} \right) = \rho_i \quad \text{for } t_i \leq t \leq t_{i+1}$$

and

$$B(t) \approx B \left(\frac{t_i + t_{i+1}}{2} \right) = B_i \quad \text{for } t_i \leq t \leq t_{i+1}.$$

Now, (3.1) becomes for $t_i \leq t \leq t_{i+1}$

$$\frac{d\vec{x}}{dt} = A\vec{x} + B_i\vec{x} + \vec{F}(t) + \hat{B}^{\frac{1}{2}} \frac{d\vec{W}}{dt}. \quad (3.6)$$

Then, using Itô's formula [7, 13],

$$\frac{d}{dt} [e^{-(A+B_i)t} \vec{x}] = e^{-(A+B_i)t} \vec{F}(t) + e^{-(A+B_i)t} \hat{B}^{\frac{1}{2}} \frac{d\vec{W}}{dt}$$

Approximating this equation using Euler's method, we derive

$$e^{-(A+B_i)t_{i+1}} \vec{x}_{i+1} = e^{-(A+B_i)t_i} \vec{x}_i + h e^{-(A+B_i)t_i} \vec{F}(t_i) + \sqrt{h} e^{-(A+B_i)t_i} \hat{B}^{\frac{1}{2}} \vec{\eta}_i.$$

Rearranging this equation gives

$$\vec{x}_{i+1} = e^{(A+B_i)h} \vec{x}_i + h e^{(A+B_i)h} \vec{F}(t_i) + \sqrt{h} e^{(A+B_i)h} \hat{B}^{\frac{1}{2}} \vec{\eta}_i. \quad (3.7)$$

Our numerical approximation is based on the above equation.

To efficiently solve (3.7), the eigenvalues and eigenvectors are rapidly computed at each time step. Let $A + B_i = X_i D_i X_i^{-1}$ where X_i are the eigenvectors of $A + B_i$ and $\omega_1^{(i)}, \omega_2^{(i)}, \dots, \omega_{m+1}^{(i)}$ are the corresponding eigenvalues of $A + B_i$. These eigenvalues are the diagonal elements of D_i [12]. Replacing $A + B_i$ with the associated decomposition, (3.7) becomes

$$\vec{x}_{i+1} = X_i e^{D_i h_i} X_i^{-1} \left\{ \left[\vec{x}_i + h \vec{F}(t_i) \right] + \sqrt{h} \widehat{B}^{\frac{1}{2}} \vec{\eta}_i \right\}. \quad (3.8)$$

Notice that after X_i and D_i are determined, (3.8) can be computed at each time step by using matrix-vector multiplications and vector additions. This ensures an efficient calculation.

For this method to be efficient, the computation of the eigenvectors and their associated eigenvalues must be efficient as well. Solving for the roots of the inhour equation provides the $m + 1$ eigenvalues of the point-kinetics matrix [9],

$$\rho_i = \beta + \ell\omega - \sum_{j=1}^m \frac{\beta_j \lambda_j}{\omega + \lambda_j} \quad (3.9)$$

which can be expressed as a polynomial $P_i(\omega)$ of degree $m + 1$,

$$P_i(\omega) = (\rho_i - \ell\omega) \prod_{l=1}^m (\lambda_l + \omega) - \omega \sum_{k=1}^m \beta_k \prod_{\substack{l=1 \\ l \neq k}}^m (\lambda_l + \omega) = 0. \quad (3.10)$$

The roots of $P_i(\omega)$ are real [9], and a modified Newton's method is efficiently used to rapidly calculate the roots of the polynomial. These results yield the set of eigenvalues, $\{\omega_1^{(i)}, \omega_2^{(i)}, \dots, \omega_{m+1}^{(i)}\}$, each of which are calculated for the i^{th} time step for $i = 0, 1, 2, \dots$.

The corresponding eigenvectors of the point-kinetics matrix have the form [9]:

$$\vec{y}_k^{(i)} = \begin{bmatrix} 1 \\ \frac{\mu_1}{\lambda_1 + \omega_k^{(i)}} \\ \frac{\mu_2}{\lambda_2 + \omega_k^{(i)}} \\ \vdots \\ \frac{\mu_m}{\lambda_m + \omega_k^{(i)}} \end{bmatrix} \quad (3.11)$$

where $\vec{y}_k^{(i)}$ is the k^{th} column of X_i for $k = 1, 2, \dots, m + 1$. Notice that $\omega_k^{(i)}$ is the associated eigenvalue and $\mu_l = \frac{\beta_l}{\ell}$ for $l = 1, 2, \dots, m$. The inverse of the matrix of eigenvectors X_i^{-1} has a similar form:

$$\vec{z}_k^{(i)} = \nu_k^{(i)} \begin{bmatrix} 1 \\ \frac{\lambda_1}{\lambda_1 + \omega_k^{(i)}} \\ \frac{\lambda_2}{\lambda_2 + \omega_k^{(i)}} \\ \vdots \\ \frac{\lambda_m}{\lambda_m + \omega_k^{(i)}} \end{bmatrix} \quad (3.12)$$

where $\vec{z}_k^{(i)}$ is the k^{th} column of X_i^{-1} and $\nu_k^{(j)} = \left[1 + \sum_{j=1}^m \frac{\mu_j \lambda_j}{(\lambda_j + \omega_k^{(i)})^2} \right]^{-1}$.

The eigenvalues and eigenvectors are computed at each time step. The matrices, X_i and D_i are therefore calculated efficiently at each time step i . Then, (3.8) is used to calculate $\vec{x}_{i+1}$, for $i = 0, 1, 2, \dots$. More information on this numerical procedure can be found in [12].

CHAPTER IV

COMPUTATIONAL RESULTS

We consider several computational examples. The stochastic model was implemented using a modified version of the PCA method described in [12]. The MATLAB code of the stochastic model is included in the Appendix. Many of the examples were also computed using Monte Carlo calculations. For simplicity, the program that models our stochastic model will be referred to as the stochastic piecewise constant approximation method, or the Stochastic PCA method. We should not expect the same results between the Monte Carlo calculations and the Stochastic PCA method; however, given the same parameters the computational results should be reasonably close.

The first two examples do not rigorously model physical nuclear reactor problems, but these problems provide simple computational solutions. These first two examples use the following parameters: $\lambda_1 = 0.1$, $\beta_1 = .05 = \beta$, $\nu = 2.5$, and a neutron source $q = 200$.

The first example simulates a step-reactivity insertion. For this case, $\ell = \frac{2}{3}$, $\rho(t) = -\frac{1}{3}$ for $t \geq 0$, and the initial condition starts at equilibrium, $\vec{x}(0) = [400, 300]^T$. The Monte Carlo calculations used 5000 trials while the Stochastic PCA method used 40 time intervals and 5000 trials. Good agreement between the two different methods when time $t = 2$ can be seen in Table 4.1 where the means of $n(2)$ and $c_1(2)$ are presented along with their standard deviations.

The second example determines the time it takes for the neutron density to reach a certain population. For this case, $\ell = 0.499002$, $\rho(t) = 0.001996$ for $t \geq 0$, and the initial condition starts at zero, $\vec{x}(0) = [0, 0]^T$. The Stochastic PCA method used a step size of $h = .05$. Computations were continued for 5000 trials for each method. The Stochastic PCA method was much faster than the Monte Carlo method for this example. The results of the cases given in Table 4.2 again indicate good agreement between the two methods.

The next two computational examples model actual step reactivity insertions in a nuclear reactor [6, 12]. The initial condition assumes a source-free equilibrium where

$$\vec{x}(0) = 100 \begin{bmatrix} 1 \\ \frac{\beta_1}{\lambda_1 \ell} \\ \frac{\beta_2}{\lambda_2 \ell} \\ \vdots \\ \frac{\beta_m}{\lambda_m \ell} \end{bmatrix}.$$

The following parameters were used in these examples: $\ell = 0.00002$, $\beta = 0.007$, $\beta_i = [0.000266, 0.001491, 0.001316, 0.002849, 0.000896, 0.000182]$, $\nu = 2.5$, $q = 0$, and $\lambda_i = [0.0127, 0.0317, 0.115, 0.311, 1.4, 3.87]$ with $m = 6$ delayed groups.

This first example models a prompt subcritical insertion, $\rho = 0.003$. Computational results at time $t = 0.1$ can be seen in Table 4.3 for the Monte Carlo and Stochastic PCA methods. Good agreement is seen between the two approaches. Using the Stochastic PCA method, the mean neutron density and two individual neutron sample paths are given in Figure 4.1, and the mean precursor density and two precursor sample paths are given in Figure 4.2. The second example uses exactly the same data but models a prompt critical insertion, $\rho = 0.007$. Computational results at time $t = 0.001$ are given in Table 4.4. For these calculations, 5000 trials were used in both the Monte Carlo calculations and the Stochastic PCA method.

The final example models the Godiva reactor [8] to determine the time it takes for the neutron level to reach 4.2×10^3 neutrons with a source of 90 neutrons/sec. For this reactor, the following parameters were used: $h = 0.1$, $\nu = 2.57$, $\beta = 0.0066$, $\beta_i = [0.00025, 0.00141, 0.00124, 0.00269, 0.00084, 0.00017]$, $\rho = 0.00462$, $\ell = 0.6 \times 10^{-8}$, $\lambda_i = [0.0127, 0.0317, 0.115, 0.311, 1.4, 3.87]$, $\vec{x}(0) = [0, 0, 0, 0, 0, 0, 0]^T$, and $q = 90$ with $m = 6$ delayed groups. These parameters were obtained from [8]. The results of 22 experimental runs [8] and the Stochastic PCA model are given in Table 4.5. The calculated times are displayed in Figure 4.3. Notice that the experimental values for the 22 experimental measurements have a mean value and a standard deviation within 5% of the values computed using the Stochastic PCA method. Displayed in

Figure 4.4 are the relative frequency histograms for the calculational results and the experimental measurements.

Table 4.1: A comparison using only one precursor

	Monte Carlo	Stochastic PCA
$E(n(2))$	400.03	395.32
$\sigma(n(2))$	27.311	29.411
$E(c_1(2))$	300.00	300.67
$\sigma(c_1(2))$	7.8073	8.3564

Table 4.2: A comparison using a given neutron level, $n_{level} = 4000$

	Monte Carlo	Stochastic PCA
$E(t)$	33.136	33.157
$\sigma(t)$	2.0886	2.5772

Table 4.3: A comparison using a prompt subcritical step reactivity, $\rho = 0.003$

	Monte Carlo	Stochastic PCA
$E(n(0.1))$	183.04	186.31
$\sigma(n(0.1))$	168.79	164.16
$E\left(\sum_{i=1}^6 c_i(0.1)\right)$	4.478×10^5	4.491×10^5
$\sigma\left(\sum_{i=1}^6 c_i(0.1)\right)$	1495.7	1917.2

Table 4.4: A comparison using prompt critical step reactivity, $\rho = 0.007$

	Monte Carlo	Stochastic PCA
$E(n(0.001))$	135.67	134.55
$\sigma(n(0.001))$	93.376	91.242
$E\left(\sum_{i=1}^6 c_i(0.001)\right)$	4.464×10^5	4.464×10^5
$\sigma\left(\sum_{i=1}^6 c_i(0.001)\right)$	16.226	19.444

Table 4.5: A comparison for a Godiva experiment

	Experimental	Stochastic PCA
$E(t)$	31.8	30.488
$\sigma(t)$	4.5826	4.7225

Figure 4.1: Neutron density using a prompt subcritical step reactivity, $\rho = 0.003$

Figure 4.2: Precursor density using a prompt subcritical step reactivity, $\rho = 0.003$

Figure 4.3: Histogram of times for the Godiva reactor to reach 4200 neutrons

Figure 4.4: Frequency histograms for the experimental measurements and calculated results for the Godiva reactor

CHAPTER V

SUMMARY AND CONCLUSIONS

The stochastic point-kinetics equations derived in the present investigation generalize the standard deterministic point-kinetics equations. In addition, the Stochastic PCA numerical method provides a fast calculational method in comparison with Monte Carlo computations.

In possible future work, the point-kinetic equations may be expanded to stochastic reactor-kinetic equations. The implementation of such a stochastic spatial model would be complicated, but the calculational results provided may be useful for studying spatially-dependent stochastic phenomena in a nuclear reactor.

BIBLIOGRAPHY

- [1] A. E. Aboanber and A. A. Nahla, Generalization of the Analytical Inversion Method for the Solution of the Point Kinetics Equations, *Journal of Physics A: Mathematical and General*, 35, 3245-3263 (2002).
- [2] A. E. Aboanber and Y. M. Hamada, Power Series Solution (PWS) of Nuclear Reactor Dynamics with Newtonian Temperature Feed Back, *Annals of Nuclear Energy*, 30, 1111-1122 (2003).
- [3] A. E. Aboanber and Y. M. Hamada, PWS: An Efficient Code System for Solving Space-Independent Nuclear Reactor Dynamics, *Annals of Nuclear Energy*, 29, 2159-2172 (2002).
- [4] E. J. Allen, Stochastic Differential Equations and Persistence Time for Two Interacting Populations, *Dynamics of Continuous, Discrete, and Impulsive Systems*, 5, 271-281 (1999).
- [5] L. J. S. Allen, *An Introduction to Stochastic Processes With Applications to Biology*, Pearson Education, Inc., Upper Saddle River, New Jersey, 2003.
- [6] Y. Chao and A. Attard, A Resolution to the Stiffness Problem of Reactor Kinetics, *Nuclear Science and Engineering*, 90, 40-46 (1985).
- [7] T. C. Gard, *Introduction to Stochastic Differential Equations*, Marcel Dekker, New York 1987.
- [8] G. E. Hansen, Assembly of Fissionable Material in the Presence of a Weak Neutron Source, *Nuclear Science and Engineering*, 8, 709-719 (1960).
- [9] D. L. Hetrick, *Dynamics of Nuclear Reactors*, The University of Chicago Press, Chicago, 1971.
- [10] H. Hurwitz, Jr., D. B. MacMillian, J. H. Smith, M. L. Storm, Kinetics of Low Source Reactor Startups Part I, *Nuclear Science and Engineering*, 15, 166-186 (1963).
- [11] H. Hurwitz, Jr., D. B. MacMillian, J. H. Smith, M. L. Storm, Kinetics of Low Source Reactor Startups Part II, *Nuclear Science and Engineering*, 15, 187-196 (1963).
- [12] M. Kinard and E. J. Allen, Efficient numerical solution of the point kinetics equations in nuclear reactor dynamics, *Annals of Nuclear Energy*, 31, 1039-1051, (2004).
- [13] P. E. Kloeden and E. Platen, *Numerical Solution of Stochastic Differential Equations*, Springer-Verlag, New York, 1992.

- [14] D. B. MacMillian and M. L. Storm, Kinetics of Low Source Reactor Startups Part III, *Nuclear Science and Engineering*, 16, 369-380 (1963).
- [15] J. Sánchez, On the Numerical Solution of the Point Kinetics Equations by Generalized Runge-Kutta Methods, *Nuclear Science and Engineering*, 103, 94-99 (1989).

APPENDIX A

STOPCA.M

It should be noted that this code is a revised version of the code given in [12]. It has been altered to include the stochastic terms.

```
function [mean_f,mean_squared,sd] = stopca(lambda,beta,beta_sum,L,
target,steps,runs,rho_case,init_cond,nu,q,l_q)
format short g
warning off

% creates mesh with equal subintervals
mesh=linspace(0,target,steps+1);

%% Determines our step size
h=target/steps;

%% Determine the number of delay groups, thereby the size of our
%% solution
m = length(lambda) + 1;

%% Creates matrices to store paths for each popultaion: neutrons,
%% precursors, and the precursor summation.
mean=zeros(steps+1,m+1);
mean_2=zeros(steps+1,m+1);
paths=zeros(steps+1,2*m+2);

%% Calculate the values of several constants that will be needed
%% in the control of the iterations as well as set up some basic
%% matrices.
x = init_cond;
time=0;
d_hat=zeros(m,m);
big_d=zeros(m,m);
```

```

iterations=steps;
result=zeros(m+1,iterations);
source=zeros(m,1);
d=zeros(m,1);
xx=3404658;

%% Stores first row with initial conditions for each value stored
%% in the vector x. It alternates with x(i) and x(i)^2 for each
%% of the values for x, then it calculates the precursor sum and
%% precursor sum squared and stores it in the first row and the
%% final two columns of each matrix.
for t=1:m
    mean(1,t)=x(t);
    mean_2(1,t)=x(t)^2;
    paths(1,2*t-1)=x(t);
    paths(1,2*t)=x(t);
end
mean(1,m+1)=precursor_sum(x);
mean_2(1,m+1)=precursor_sum(x)^2;
paths(1,2*m+1)=precursor_sum(x);
paths(1,2*m+2)=precursor_sum(x);

%% Begins algorithm
for jj=1:runs
    x=init_cond;
    for i=1:steps
        time=time+h;
        %% Calculate the values of the reactivity and source at the
        %% midpoint
        mid_time=time-(h/2);
        p=rho(rho_case,beta_sum,mid_time);

        %% Caculate the roots to the inhour equation

```

```

d=inhour(lambda,L,beta,p,d);

%% Calculate the eigenvectors and the inverse of the matrix
%% of eigenvectors
Y=ev2(lambda,L,beta,d);
Y_inv=ev_inv(lambda,L,beta,d);

%% Construct matrices for computation
for k=1:m
    d_hat(k,k)=exp(d(k)*h);
    big_d(k,k)=d(k);
end

big_d_inv=zeros(m,m);

for k=1:m
    big_d_inv(k,k)=1/big_d(k,k);
end

%% Creates random vector and solves for next iteration
rand=random7(xx);
xx=rand(8);
for k=1:m
    ran(k,1)=rand(k);
end
B_hat=B_hat_Mat(x,lambda,beta,beta_sum,L,p,nu,q,l_q);
source(1)=q*l_q*h;
x=(Y*d_hat*Y_inv)*(x+sqrt(h)*((B_hat)^(1/2))*ran+source);

for t=1:length(x)
    if x(t)<0
        x(t)=0;
    end
end

```



```

end

%% Store results in a matrix
for j=1:m
    result(1,i)=i*h;
    result(j+1,i)=x(j);
    mean(i+1,j)=mean(i+1,j)+x(j)/runs;
    mean_2(i+1,j)=mean_2(i+1,j)+x(j)^2/runs;
    paths(i+1,2*j)=paths(i+1,2*j-1);
    paths(i+1,2*j-1)=x(j);
end
mean(i+1,m+1)=mean(i+1,m+1)+precursor_sum(x)/runs;
mean_2(i+1,m+1)=mean_2(i+1,m+1)+precursor_sum(x)^2/runs;
paths(i+1,2*m+2)=paths(i+1,2*m+1);
paths(i+1,2*m+1)=precursor_sum(x);
end
end

%% Finds mean of neutrons, precursors, and precursor summation.
mean_f=mean(steps+1,:);

%% Finds the mean squared of neutrons, precursors, and precursor
%% summation.
mean_squared=mean_2(steps+1,:);

%% Finds the variance of neutrons, precursors, and precursor
%% summation.
for r=1:m+1
    sd(r,1)=sqrt(mean_squared(r,1)-mean_f(r,1)^2);
end

%% Graphical data
figure(1)

```

```

plot(mesh,paths(:,1),'b',mesh,paths(:,2),'g',mesh,mean(:,1),'r',mesh,
paths(:,2*m+1),'c',mesh,paths(:,2*m+2),'m',mesh,mean(:,m+1),'k')
xlabel('Time')
ylabel('Population')
figure(2)
plot(mesh,paths(:,2*m+1),'--b',mesh,paths(:,2*m+2),'-.g',mesh,
mean(:,m+1),'r')
xlabel('Time')
ylabel('Sum of Neutron Precursors Population')
legend('Sample Precursor','Sample Precursor','Precursor Mean')
figure(3)
plot(mesh,paths(:,1),'--b',mesh,paths(:,2),'-.g',mesh,mean(:,1),'r')
xlabel('Time')
ylabel('Neutron Population')
legend('Sample Neutron','Sample Neutron','Neutron Mean')
%-----
function y=ev2(lambda,L,beta,evals)
%%This is a simple function that calculates the eigenvectors using
%% the appropriate forms.

m=length(lambda) + 1;
evecs=zeros(m,m);
for i=1:m
    for j=1:m
        if i==1
            evecs(i,j) = 1;
        end
        if i~=1
            mu=beta(i-1)/L;
            evecs(i,j)=mu/(lambda(i-1)+evals(j));
        end
    end
end
end

```

```

y=evects;
%-----
function y=ev_inv(lambda,L,beta,evals)
%% This function returns the inverse of the matrix of eigenvalues
%% based on some computations provided in Aboanber and Nahla.

m=length(lambda)+1;
for i=1:m-1
    mu(i)=beta(i)/L;
end

normfact=zeros(m,1);

for k=1:m
    sum=0;
    for i=1:m-1
        temp=mu(i)*lambda(i);
        temp2=(lambda(i)+evals(k))^2;
        temp3=temp/temp2;
        sum=sum+temp3;
    end
    normfact(k)=1/(sum+1);
end

result=zeros(m,m);
for i=1:m
    for j=1:m
        if i==1
            result(i,j)=1*normfact(j);
        end
        if i~=1
            result(i,j)=(lambda(i-1)/(lambda(i-1)+evals(j)))*normfact(j);
        end
    end
end

```

```

        end
    end
    y=transpose(result);
    %-----
    function y=expand(lambda)
    %% A simple helper function to provide the coefficients of a
    %% polynomial produced by raising the function (x+y) to the nth power.
    %% The argument, lambda is a vector of constants that are needed to
    %% derive the coefficients.

    %% Determines the number of iterations, as well as the degree
    %% of the polynomial in question
    m=length(lambda);

    coeff=zeros(m+1,1);

    %% A temporary variable is necessary b/c the iterations that follow
    %% require information from the previous iteration...

    temp=coeff;

    %% Must run the index to m+1 b/c MATLAB uses a 1-based index
    for i=1:m+1
        if i~=1
            coeff(1)=temp(1)*lambda(i-1);
            for j=2:m+1
                coeff(j)=temp(j)*lambda(i-1)+temp(j-1);
                if j==i-1
                    coeff(j)=temp(j-1)+lambda(i-1);
                end
            end
        end
        end
        coeff(i)=1;

```

```

        temp=coeff;
    end
    y=coeff;
    %-----
    function y=inhour(lambda,L,beta,rho,init_root)
    %% This function begins by taking the arguments and converting them
    %% into the correct m-degree polynomial inorder to take advantage
    %% of the given method of finding the roots of said polynomial.

    m=length(lambda);
    sum=zeros(m,1);
    coeff=expand(lambda);
    coeff_2=zeros(m+2,1);

    for i=2:m+1
        coeff_2(i)=rho*coeff(i)-L*coeff(i-1);
    end

    coeff_2(1)=rho*coeff(1);
    coeff_2(m+2)=-L*coeff(m+1);

    for i=1:m
        temp_lambda=trunc(lambda,i);
        temp=beta(i)*expand(temp_lambda);
        sum=temp + sum;
    end

    sum=-1*sum;
    res=zeros(m+2,1);
    for i=1:m
        res(i+1)=coeff_2(i+1)+sum(i);
    end
end

```

```

res(1)=coeff_2(1);
res(m+2)=coeff_2(m+2);

e_vals=rootfinder(res,init_root,.00001);
y=e_vals;
%-----
function y=myDeriv(coeff)
%% A simple function that calculates the derivative coefficient
%% vector for a given polynomial.

deg=length(coeff);
if deg~=1
    result=zeros(1,deg-1);
    for i=1:(deg-1)
        result(i)=coeff(i+1)*i;
    end
end
if deg==1
    result=0;
end
y=result;
%-----
function y=myEval(coeff,x)
%% Evaluates the polynomial expressed as coeff at the value x.

deg=length(coeff);
sum=coeff(1);
if deg~=1
    for i=2:deg
        sum=sum+coeff(i)*x^(i-1);
    end
end
y=sum;

```

```

%-----
function y=myHorner(a,z,n)
%%Applies a functional implementation of the Horner method
%%The user supplies a(The poly), z(The root), and n(The degree)
%%This program uses Horner's method to write  $p(x) = (x-z)q(x)+c$ 
%%Where p and q are polynomials of degree n and n-1 respectively

for i=1:n+1
    b(i)=0.0;
end

b(n)=a(n+1);

if n>0
    for i=1:n
        b(n+1-i)=a(n-i+2)+b(n+2-i)*z;
    end
    c=a(1)+b(1)*z;
end
for i=1:n
    a(i)=b(i);
end
for i=1:n
    ret(i)=a(i);
end
ret(n)=ret(n)+c; %%add the constant
y=ret;
%-----

function y=newton(val,poly,tol)
%% A simple implementation of Newton's Method

eps=1;
x=val;

```

```

deriv=myDeriv(poly);
while eps>tol
    temp=x-(myEval(poly,x)/myEval(deriv,x));
    eps=abs(x-temp);
    x=temp;
end
y=x;
%-----
function y=rho(case_number,beta_sum,t)
%% This function represents the time-dependent reactivity function
%% for the point kinetics equation. We will use the argument "case"
%% to determine what type of reactivity we have in question.

%% case_number = 1 : Step reactivity of rho = -1/3
%% case_number = 2 : Step reactivity of rho = 0.001996
%% case_number = 3 : Step reactivity of rho = 0.003
%% case_number = 4 : Step reactivity of rho = 0.007
%% case_number = 5 : Step reactivity of rho = 0
%% case_number = 6 : Step reactivity of rho = 0.00462

if case_number==1
    result=-1/3;
end
if case_number==2
    result=0.001996;
end
if case_number==3
    result=0.003;
end
if case_number==4
    result=0.007;
end
if case_number==5

```



```

        result=0;
    end
    if case_number==6
        result=0.00462;
    end
    y=result;
    %-----
    function y=rootfinder(coeff, init, tol)
    %% This is a simple wrapper function that takes an coefficient
    %% vector and uses Newton's method to find all of the real roots of
    %% said poly. The function takes advantage of Horner's method to
    %% deflate the poly at each step to expedite computation. The
    %% argument init is a vector of initial values that are used in
    %% Newton's method.

    deg=length(coeff)-1;
    result=zeros(deg,1);
    counter=1;

    while deg>1
        result(counter)=newton(init(counter),coeff,tol);
        coeff=myHorner(coeff,result(counter),deg) ;
        deg=deg-1;
        counter=counter+1;
    end
    result(counter)=-coeff(1)/coeff(2);
    y=result;
    %-----
    function y=swap(arg)
    %% This simple function takes the 1st element of arg and puts it in
    %% the mth place of the resultant vector, and puts the 2nd in the
    %% m-1st...and so on...

```

```

m=length(arg);
res=zeros(1,m);
for i=1:m
    res(m-i+1)=arg(i);
end
y=res;
%-----
function y=trunc(var,t)
%% This is a simple helper method that removes the ith element
%% from the vector var.

m=length(var);
flag=0;
temp=zeros(1,m-1);
for i=1:m
    if i~=t & flag==0
        temp(i)=var(i);
    end
    if i~=t & flag==1
        temp(i-1)=var(i);
    end
    if i==t
        flag=1;
    end
end
y=transpose(temp);
%-----
%% Random Number Generator
%% Multiplicative congruential generator  $xx=16807*xx \bmod (2^{31}-1)$ 
%% Box-Muller method converts to normal random numbers
%%  $xx=rand(8)$  is input to the generator

function rand=random7(xx)

```

```

a=16807;
b=2147483647;
for i=1:8
    d=fix(a*xx/b);
    xx=a*xx-d*b;
    rng(i)=xx/b;
end
p=3.141592654;
for j=1:4
    u1=rng(2*j-1);
    u2=rng(2*j);
    hlp=sqrt(-2.0*log(u1));
    rand(2*j-1)=hlp*cos(p*2.0*u2);
    rand(2*j)=hlp*sin(p*2.0*u2);
    rand(8)=xx;
end
%-----
%% Precursor Sum
%% This program adds up the population of each precursor

function z=precursor_sum(y)
t=0;
for i=2:length(y)
    t=t+y(i);
end
z=t;
%-----
%% Precursor Source Summation
%% This program determines  $\Sigma(\lambda_i * c_i)$ 

function z=precursor_source(lambda,y)
z=0;
for i=2:length(y)

```

```

        z=z+lambda(i-1)*y(i);
    end
    %-----
    %% B_hat_Mat
    %% This program creates our stochastic matrix B_hat

    function z=B_hat_Mat(y,lambda,beta,beta_sum,L,rho,nu,q,l_q)
    B_hat=zeros(length(lambda)+1,length(lambda)+1);
    b=zeros(length(lambda)+1,length(lambda)+1);
    alpha=1/nu;
    birth=(1-alpha-beta_sum)/L;
    death=(-rho+1-alpha)/L;
    source_c=precursor_source(lambda,y);
    n=(death*y(1))+(birth*y(1)*(-1+(1-beta_sum)*nu))+source_c+(q*l_q);
    for k=1:length(lambda)
        c(k)=(((beta(k)^2)*nu)/L)*y(1)+(lambda(k)*y(k+1));
    end
    for k=1:length(lambda)
        a(k)=((beta(k)/L)*(-1+(1-beta_sum)*nu)*y(1))-(lambda(k)*y(k+1));
    end
    for i=2:length(lambda)
        for j=2:length(lambda)
            if i~=j
                b(i,j)=(beta(i-1)*beta(j-1)*nu*y(1))/L;
            end
        end
    end
    end
    for k=1:(length(lambda)+1)
        for j=1:(length(lambda)+1)
            if k==j
                if k==1
                    B_hat(k,j)=n;
                else

```

```

        B_hat(k,j)=c(k-1);
    end
elseif k==1 && j>1
    B_hat(k,j)=a(j-1);
elseif k>1 && j==1
    B_hat(k,j)=a(k-1);
else
    B_hat(k,j)=b(k,j);
end
end
end
z=B_hat;

```

APPENDIX B

STOHIST.M

It should be noted that this code is a revised version of the code given in [12]. It has been altered to include the stochastic terms.

```
function [tta,tt2,sd,data]=stohist(lambda,beta,beta_sum,L,h,runs,
rho_case,init_cond,nu,q,l_q,level)
format short g
warning off MATLAB:sqrtm:SingularMatrix

%% Determine the number of delay groups, thereby the size of our
%% solution
m=length(lambda)+1;

%% Calculate the values of several constants that will be needed in
%% the control of the iterations as well as set up some basic matrices.
x=init_cond;
d_hat=zeros(m,m);
big_d=zeros(m,m);
source=zeros(m,1);
ran=zeros(m,1);
d=zeros(m,1);
data=zeros(runs,1);
tt2=0;
tta=0;
xx=3404658;

%% Begin time dependent iterations
for ica=1:runs
    x=init_cond;
    time=-h;
    while x(1)<=level
        time=time+h;
```

```

%% Calculate the values of the reactivity and source at
%% the midpoint
mid_time=time-(h/2);
p=rho(rho_case,beta_sum,mid_time);

%% Caculate the roots to the inhour equation
d=inhour(lambda,L,beta,p,d);

%% Calculate the eigenvectors and the inverse of the matrix
%% of eigenvectors
Y=ev2(lambda,L,beta,d);
Y_inv=ev_inv(lambda,L,beta,d);

%% Construct matrices for computation
for k=1:m
    d_hat(k,k)=exp(d(k)*h);
    big_d(k,k)=d(k);
end

big_d_inv=zeros(m,m);

for k=1:m
    big_d_inv(k,k)=1/big_d(k,k);
end

rand=random7(xx);
xx=rand(8);
for i=1:m
    ran(i,1)=rand(i);
end
B_hat=B_hat_Mat(x,lambda,beta,beta_sum,L,p,nu,q,l_q);
source(1)=q*l_q*h;
x=(Y*d_hat*Y_inv)*(x+sqrt(h)*((B_hat)^(1/2))*ran+source);

```

```

        for t=1:length(x)
            if x(t)<0
                x(t)=0;
            end
        end
        data(icasel,1)=time;
    end
    tta=tta+data(icasel,1)/runs;
    tt2=tt2+data(icasel,1)^2/runs;
end
sd=sqrt(tt2-tta^2);

%% Graphical results
hist(data,runs);
xlabel('Time');
ylabel('Occurrences');
title(['Histogram Plot of Neutron Population Reaching ',
int2str(level),' Neutrons']);
%-----
function y=ev2(lambda,L,beta,evals)
%%This is a simple function that calculates the eigenvectors using
%% the appropriate forms.

m=length(lambda) + 1;
evecs=zeros(m,m);
for i=1:m
    for j=1:m
        if i==1
            evecs(i,j) = 1;
        end
        if i~=1
            mu=beta(i-1)/L;

```



```

            evecs(i,j)=mu/(lambda(i-1)+evals(j));
        end
    end
end
y=evecs;
%-----
function y=ev_inv(lambda,L,beta,evals)
%% This function returns the inverse of the matrix of eigenvalues
%% based on some computations provided in Aboanber and Nahla.

m=length(lambda)+1;
for i=1:m-1
    mu(i)=beta(i)/L;
end

normfact=zeros(m,1);

for k=1:m
    sum=0;
    for i=1:m-1
        temp=mu(i)*lambda(i);
        temp2=(lambda(i)+evals(k))^2;
        temp3=temp/temp2;
        sum=sum+temp3;
    end
    normfact(k)=1/(sum+1);
end

result=zeros(m,m);
for i=1:m
    for j=1:m
        if i==1
            result(i,j)=1*normfact(j);

```

```

        end
        if i~=1
            result(i,j)=(lambda(i-1)/(lambda(i-1)+evals(j)))*normfact(j);
        end
    end
end
end
y=transpose(result);
%-----
function y=expand(lambda)
%% A simple helper function to provide the coefficients of a
%% polynomial produced by raising the function (x+y) to the nth power.
%% The argument, lambda is a vector of constants that are needed to
%% derive the coefficients.

%% Determines the number of iterations, as well as the degree
%% of the polynomial in question
m=length(lambda);

coeff=zeros(m+1,1);

%% A temporary variable is necessary b/c the iterations that follow
%% require information from the previous iteration...

temp=coeff;

%% Must run the index to m+1 b/c MATLAB uses a 1-based index
for i=1:m+1
    if i~=1
        coeff(1)=temp(1)*lambda(i-1);
        for j=2:m+1
            coeff(j)=temp(j)*lambda(i-1)+temp(j-1);
            if j==i-1
                coeff(j)=temp(j-1)+lambda(i-1);
            end
        end
    end
end

```

```

        end
    end
end
coeff(i)=1;
temp=coeff;
end
y=coeff;
%-----
function y=inhour(lambda,L,beta,rho,init_root)
%% This function begins by taking the arguments and converting them
%% into the correct m-degree polynomial inorder to take advantage
%% of the given method of finding the roots of said polynomial.

m=length(lambda);
sum=zeros(m,1);
coeff=expand(lambda);
coeff_2=zeros(m+2,1);

for i=2:m+1
    coeff_2(i)=rho*coeff(i)-L*coeff(i-1);
end

coeff_2(1)=rho*coeff(1);
coeff_2(m+2)=-L*coeff(m+1);

for i=1:m
    temp_lambda=trunc(lambda,i);
    temp=beta(i)*expand(temp_lambda);
    sum=temp + sum;
end

sum=-1*sum;
res=zeros(m+2,1);

```

```

for i=1:m
    res(i+1)=coeff_2(i+1)+sum(i);
end

res(1)=coeff_2(1);
res(m+2)=coeff_2(m+2);

e_vals=rootfinder(res,init_root,.00001);
y=e_vals;
%-----
function y=myDeriv(coeff)
%% A simple function that calculates the derivative coefficient
%% vector for a given polynomial.

deg=length(coeff);
if deg~=1
    result=zeros(1,deg-1);
    for i=1:(deg-1)
        result(i)=coeff(i+1)*i;
    end
end
if deg==1
    result=0;
end
y=result;
%-----
function y=myEval(coeff,x)
%% Evaluates the polynomial expressed as coeff at the value x.

deg=length(coeff);
sum=coeff(1);
if deg~=1
    for i=2:deg

```

```

        sum=sum+coeff(i)*x^(i-1);
    end
end
y=sum;
%-----
function y=myHorner(a,z,n)
%%Applies a functional implementation of the Horner method
%%The user supplies a(The poly), z(The root), and n(The degree)
%%This program uses Horner's method to write  $p(x) = (x-z)q(x)+c$ 
%%Where p and q are polynomials of degree n and n-1 respectively

for i=1:n+1
    b(i)=0.0;
end

b(n)=a(n+1);

if n>0
    for i=1:n
        b(n+1-i)=a(n-i+2)+b(n+2-i)*z;
    end
    c=a(1)+b(1)*z;
end
for i=1:n
    a(i)=b(i);
end
for i=1:n
    ret(i)=a(i);
end
ret(n)=ret(n)+c; %%add the constant
y=ret;
%-----
function y=newton(val,poly,tol)

```

```

%% A simple implementation of Newton's Method

eps=1;
x=val;
deriv=myDeriv(poly);
while eps>tol
    temp=x-(myEval(poly,x)/myEval(deriv,x));
    eps=abs(x-temp);
    x=temp;
end
y=x;
%-----
function y=rho(case_number,beta_sum,t)
%% This function represents the time-dependent reactivity function
%% for the point kinetics equation. We will use the argument "case"
%% to determine what type of reactivity we have in question.

%% case_number = 1 : Step reactivity of rho = -1/3
%% case_number = 2 : Step reactivity of rho = 0.001996
%% case_number = 3 : Step reactivity of rho = 0.003
%% case_number = 4 : Step reactivity of rho = 0.007
%% case_number = 5 : Step reactivity of rho = 0
%% case_number = 6 : Step reactivity of rho = 0.00462

if case_number==1
    result=-1/3;
end
if case_number==2
    result=0.001996;
end
if case_number==3
    result=0.003;
end
end

```

```

if case_number==4
    result=0.007;
end
if case_number==5
    result=0;
end
if case_number==6
    result=0.00462;
end
y=result;
%-----
function y=rootfinder(coeff, init, tol)
%% This is a simple wrapper function that takes an coefficient
%% vector and uses Newton's method to find all of the real roots of
%% said poly. The function takes advantage of Horner's method to
%% deflate the poly at each step to expedite computation. The
%% argument init is a vector of initial values that are used in
%% Newton's method.

deg=length(coeff)-1;
result=zeros(deg,1);
counter=1;

while deg>1
    result(counter)=newton(init(counter),coeff,tol);
    coeff=myHorner(coeff,result(counter),deg) ;
    deg=deg-1;
    counter=counter+1;
end
result(counter)=-coeff(1)/coeff(2);
y=result;
%-----
function y=swap(arg)

```

```

%% This simple function takes the 1st element of arg and puts it in
%% the mth place of the resultant vector, and puts the 2nd in the
%% m-1st...and so on...

m=length(arg);
res=zeros(1,m);
for i=1:m
    res(m-i+1)=arg(i);
end
y=res;
%-----
function y=trunc(var,t)
%% This is a simple helper method that removes the ith element
%% from the vector var.

m=length(var);
flag=0;
temp=zeros(1,m-1);
for i=1:m
    if i~=t & flag==0
        temp(i)=var(i);
    end
    if i~=t & flag==1
        temp(i-1)=var(i);
    end
    if i==t
        flag=1;
    end
end
y=transpose(temp);
%-----
%% Random Number Generator
%% Multiplicative congruential generator xx=16807*xx mod(2^31-1)

```



```

%% Box-Muller method converts to normal random numbers
%% xx=rand(8) is input to the generator

function rand=random7(xx)
a=16807;
b=2147483647;
for i=1:8
    d=fix(a*xx/b);
    xx=a*xx-d*b;
    rng(i)=xx/b;
end
p=3.141592654;
for j=1:4
    u1=rng(2*j-1);
    u2=rng(2*j);
    hlp=sqrt(-2.0*log(u1));
    rand(2*j-1)=hlp*cos(p*2.0*u2);
    rand(2*j)=hlp*sin(p*2.0*u2);
    rand(8)=xx;
end
%-----
%% Precursor Sum
%% This program adds up the population of each precursor

function z=precursor_sum(y)
t=0;
for i=2:length(y)
    t=t+y(i);
end
z=t;
%-----
%% Precursor Source Summation
%% This program determines  $\Sigma(\text{Lamba}_i * c_i)$ 

```

```

function z=precursor_source(lambda,y)
z=0;
for i=2:length(y)
    z=z+lambda(i-1)*y(i);
end
%-----
%% B_hat_Mat
%% This program creates our stochastic matrix B_hat

function z=B_hat_Mat(y,lambda,beta,beta_sum,L,rho,nu,q,l_q)
B_hat=zeros(length(lambda)+1,length(lambda)+1);
b=zeros(length(lambda)+1,length(lambda)+1);
alpha=1/nu;
birth=(1-alpha-beta_sum)/L;
death=(-rho+1-alpha)/L;
source_c=precursor_source(lambda,y);
n=(death*y(1))+(birth*y(1)*(-1+(1-beta_sum)*nu))+source_c+(q*l_q);
for k=1:length(lambda)
    c(k)=(((beta(k)^2)*nu)/L)*y(1)+(lambda(k)*y(k+1));
end
for k=1:length(lambda)
    a(k)=((beta(k)/L)*(-1+(1-beta_sum)*nu)*y(1))-(lambda(k)*y(k+1));
end
for i=2:length(lambda)
    for j=2:length(lambda)
        if i~=j
            b(i,j)=(beta(i-1)*beta(j-1)*nu*y(1))/L;
        end
    end
end
end
for k=1:(length(lambda)+1)
    for j=1:(length(lambda)+1)

```

```

    if k==j
        if k==1
            B_hat(k,j)=n;
        else
            B_hat(k,j)=c(k-1);
        end
    elseif k==1 && j>1
        B_hat(k,j)=a(j-1);
    elseif k>1 && j==1
        B_hat(k,j)=a(k-1);
    else
        B_hat(k,j)=b(k,j);
    end
end
end
z=B_hat;

```

PERMISSION TO COPY

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Texas Tech University or Texas Tech University Health Sciences Center, I agree that the Library and my major department shall make it freely available for research purposes. Permission to copy this thesis for scholarly purposes may be granted by the Director of the Library or my major professor. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my further written permission and that any user may be liable for copyright infringement.

Agree (Permission is granted.)

James G. Hayes
Student Signature

05/02/05
Date

Disagree (Permission is not granted.)

Student Signature

Date