ACKNOWLEDGEMENTS

I would first like to thank Dr. Edward Allen, and all of the mathematics faculty at Texas Tech University for all of their help and kindness in this process. The mathematics department has a reputation as one who treats their graduate students with the utmost respect, and they certainly live up to that reputation. To my amigos here at Tech, you guys are the best. No one could get through all of this grad school mess alone, I owe a great debt of gratitude to all of you! I would also like to think the mathematics and computer science faculty at Austin College for encouraging me to go to graduate school, it was one of the best decisions that I have ever made.

I can't forget my family. Mom, Dad, and Erin, you have always been there for me, and I appreciate it more than you probably know. You have all been amazing academic role models, and you have never hesitated to support whatever wacky stuff I might dream up. I also need thank my wife, Sarah. She has been extremely supportive of my efforts here at Tech, even though Lubbock was never her first choice. I could not imagine trying to go through this program without her incredible love and support. And finally, I have to say thanks to the great mystery, you are the one who makes all things possible.

CONTENTS

ABSTRACT

A numerical procedure to efficiently calculate the solution to the point kinetics equation in nuclear reactor dynamics is described and investigated. Piecewise constant approximations of the reactivity and source functions are made. The resulting system of linear differential equations is solved exactly over each time step. The method is proved to converge with order h^2 where h is the time step. The procedure is tested using a variety of initial conditions, data, and reactivity functions. The computational results indicate that the method is efficient and accurate.

LIST OF TABLES

LIST OF FIGURES

CHAPTER I
INTRODUCTION

The point kinetics equations in nuclear reactor dynamics are a system of coupled linear ordinary differential equations. The time-dependent parameters in this system are the reactivity and the neutron source terms. In the proposed numerical method, these time-dependent functions are approximated by piecewise constant functions over a partition of the total time interval. The resulting system of differential equations is then solved exactly over each time step in the partition.

An important property of the point kinetics equations is the stiffness of the system. It is well known that stiffness is a severe problem in numerical solution of the point kinetics equations and results in the need for small time steps in a computational scheme. However, if the reactivity and neutron source functions are slowly varying with respect to time, then piecewise constant approximations to these functions are accurate. The piecewise constant assumption allows us to solve the point kinetics equation exactly, and the problem of stiffness disappears. One of the advantages of the numerical method explored here is its simplicity. Furthermore, the method is efficient and accurate.

There has been a great deal of research done in the field of reactor kinetics. Much of the work has focused on eliminating the aforementioned problem of stiffness. Chao and Attard investigated this problem and developed the "stiffness confinement method" that places emphasis on the physical analogies of the point kinetics equations in order to deal with the problem [5]. Sánchez devised and implemented an A-Stable Runge-Kutta Method to avoid the problem of stiffness [3]. Several other creative schemes have been implemented as well. J.A.W. da Nórbrega uses a Padé approximation to the solution [4], and Aboanber and Nahla [1] derived a technique based on the analytical inversion of polynomials to aid in the solution to these equations.

An attractive feature of the computational method proposed in the present investigation is its simplicity. This feature makes the method easy to program and

understand. In addition, the computational method described here is accurate and efficient.

This paper is organized in the following manner. First, a basic formulation of the point kinetics equations is given along with a derivation of the solution. Then, the numerical scheme is described. Section 3 deals with the computational results using the new numerical method. An error analysis of the method is presented in section 4. Section 5 summarizes the results of the present investigation and concluding remarks are made.

CHAPTER II

NUMERICAL METHOD

The point kinetics equations for m delayed groups are [2]:

$$\frac{dn(t)}{dt} = \frac{\rho(t) - \beta}{\Lambda} n(t) + \sum_{i=1}^{m} \lambda_i C_i(t) + F(t). \tag{2.1}$$

$$\frac{C_i(t)}{dt} = \frac{\beta_i}{\Lambda} n(t) - \lambda_i C_i(t) \qquad i = 1, 2, ..., m \tag{2.2}$$

where $n(t)$ is the time-dependent neutron density, $C_i(t)$ is the i^{th} precursor density, $\rho(t)$ is the time-dependent reactivity function, β_i is the i^{th} delayed fraction, and $\beta = \sum_{i=1}^{m} \beta_i$ is the total delayed fraction. In addition, Λ is the neutron generation time, λ_i is the i^{th}-group decay constant, and $F(t)$ is the time-dependent neutron source function.

A useful way to consider this problem is by putting the problem in matrix form. In the present investigation, the point kinetics equations is considered in the form:

$$\frac{d\vec{x}}{dt} = A\vec{x} + B(t)\vec{x} + \vec{F}(t) \tag{2.3}$$

$$\vec{x}(0) = \vec{x}_0$$

where we define A as the $(m+1) \times (m+1)$ matrix:

$$A = \begin{pmatrix} \frac{-\beta}{\Lambda} & \lambda_1 & \lambda_2 & \cdots & \lambda_m \\ \frac{\beta_1}{\Lambda} & -\lambda_1 & 0 & \cdots & 0 \\ \frac{\beta_2}{\Lambda} & 0 & -\lambda_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\beta_m}{\Lambda} & 0 & 0 & \cdots & -\lambda_m \end{pmatrix},$$

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

$$B(t) = \begin{pmatrix} \frac{\rho(t)}{\Lambda} & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}$$

and, $\vec{F}(t)$ is defined as:

$$\vec{F}(t) = \begin{pmatrix} F(t) \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

It is often the case that the reactivity and source functions are slowly varying with respect to time, in comparison with the solution of equation (2.3). The numerical method approximates these two functions by piecewise constant functions over a partition in time. Specifically, let

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

and

$$\vec{F}(t) \approx \vec{F}(\frac{t_i + t_{i+1}}{2}) = \vec{F}_i \quad \text{for} \quad t_i \leq t \leq t_{i+1}$$

yielding the following initial-value problem:

$$\frac{d\hat{\vec{x}}}{dt} = A\hat{\vec{x}} + B_i\hat{\vec{x}} + \vec{F}_i \tag{2.4}$$
$$\hat{\vec{x}}(t_i) = \hat{\vec{x}}_i$$

where $\hat{\vec{x}}(t) \approx \vec{x}(t)$

Notice that equation (2.4) can now be solved exactly. Multiplying both sides by the integrating factor $e^{-(A+B_i)t}$ yields:

$$\frac{d}{dt}(e^{-(A+B_i)t} Hat\vec{x}) = e^{-(A+B_i)t}\vec{F}_i. \tag{2.5}$$

Integrating both sides of the equation over the i^{th} time step from t_i to t_{i+1} gives,

$$e^{-(A+B_i)t_{i+1}}\hat{\vec{x}}_{i+1} - e^{-(A+B_i)t_i}\hat{\vec{x}}_i = \int_{t_i}^{t_{i+1}} e^{-(A+B_i)u}\vec{F}_i du. \tag{2.6}$$

Subtracting and evaluating the integral yields an equation of the form:

$$\hat{\vec{x}}_{i+1} = e^{(A+B_i)h_i}\hat{\vec{x}}_i + (e^{(A+B_i)h_i} - I)(A+B_i)^{-1}\vec{F}_i \quad \text{where} \quad h_i = t_{i+1} - t_i \tag{2.7}$$

The numerical scheme proposed in the present investigation is based on this equation. We now consider an efficient way to compute this equation.

Let $A + B_i = X_i D_i X_i^{-1}$ where X_i are the eigenvectors of $A + B_i$ and $D_i = \text{diag}[\omega_1^{(i)}, \cdots, \omega_{m+1}^{(i)}]$. The values $\omega_1^{(i)}, \omega_2^{(i)}, \cdots, \omega_{m+1}^{(i)}$ are the associated eigenvalues of $A + B_i$ (Note: a discussion of the efficient calculation of the eigenvalues and eigenvectors is given later). Replacing $A + B_i$ with the associated decomposition, equation (2.7) becomes:

$$\hat{\vec{x}}_{i+1} = X_i e^{D_i h_i} X_i^{-1}\hat{\vec{x}}_i + (X_i e^{D_i h_i} X_i^{-1} - I) X_i D_i^{-1} X_i^{-1}\vec{F}_i \tag{2.8}$$

or equivalently,

$$\hat{\vec{x}}_{i+1} = X_i e^{D_i h_i} X_i^{-1}[\hat{\vec{x}}_i + X_i D_i^{-1} X_i^{-1}\vec{F}_i] - X_i D_i^{-1} X_i^{-1}\vec{F}_i \tag{2.9}$$

Notice that after X_i and D_i are determined, (2.9) can be computed at each time step by using only a series of matrix-vector multiplications and vector additions. Hence (2.9) can be computed rapidly once X_i and D_i are calculated.

Thus, in order for the method to be efficient, the computation of the eigenvectors and their associated eigenvalues must be efficient as well. It is well known that the $(m+1)$ eigenvalues of the point-kinetics matrix are the roots of the inhour equation [2]:

$$\rho_i = \beta + \Lambda\omega - \sum_{j=1}^{m} \frac{\beta_j \lambda_j}{\omega + \lambda_j} \tag{2.10}$$

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

$$P_i(\omega) = (\rho_i - \omega\Lambda) \prod_{l=1}^{m} (\lambda_l + \omega) - \omega \sum_{k=1}^{m} \beta_k \prod_{\substack{l=1 \\ l \neq k}}^{m} (\lambda_l + \omega) = 0 \tag{2.11}$$

Note that it is well-known that the roots of $P_i(\omega)$ are real [2]. A modified Newton's method is used to calculate the roots of the polynomial. After a given root, ω_i, is calculated using Newton's method, the polynomial is deflated using Horner's method. (All of the algorithms are provided as MATLAB codes in Appendix A.) These calculations yield the set of eigenvalues $\{\omega_1^{(i)}, \omega_2^{(i)}, \cdots, \omega_{m+1}^{(i)}\}$, each calculated for the i^{th} time step. This procedure is very rapid and calculating each eigenvalue only requires three or four iterations of Newton's method to yield accurate results.

The task of calculating the eigenvectors is quite simple. Hetrick [2] points out that the eigenvectors of the point-kinetics matrix have the form

$$X_i = U_i \qquad \text{where} \qquad \vec{u}_k^{(i)} = \left[1, \frac{\mu_1}{\lambda_1 + \omega_k^{(i)}}, \frac{\mu_2}{\lambda_2 + \omega_k^{(i)}}, \cdots, \frac{\mu_m}{\lambda_m + \omega_k^{(i)}}\right]^T$$

and where each $\vec{u}_k^{(i)}$ is the kth column of U_i for $k = 1, 2, \ldots, m+1$. Note that the $\omega_k^{(i)}$ are the appropriate eigenvalues, and $\mu_l = \frac{\beta_l}{\Lambda}$ for $l = 1, 2, \ldots, m$. In addition, the inverse of the matrix of eigenvectors can be written as:

$$X_i^{-1} = V_i^T \qquad \text{where} \qquad \vec{v}_k^{(i)} = \nu_k^{(i)} \left[1, \frac{\lambda_1}{\lambda_1 + \omega_k^{(i)}}, \frac{\lambda_2}{\lambda_2 + \omega_k^{(i)}}, \cdots, \frac{\lambda_m}{\lambda_m + \omega_k^{(i)}}\right]^T$$

is the kth column of V_i 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 i if the reactivity and source functions change with time. Thus, the matrices, X_i and D_i are efficiently calculated at each step i and equation (2.9) is used to calculate $\vec{x}_{i+1}$. We now consider computational results obtained from this method.

CHAPTER III
ANALYSIS AND COMPUTATIONAL RESULTS

We consider a variety of computational examples. All of the examples were computed using the included MATLAB code. Built-in MATLAB functions were not used so that other investigators can easily translate the MATLAB code into another programming language such as FORTRAN or C++. Not using the built-in functions also gives us complete control over the method. For simplicity, throughout the remainder of the paper, the proposed method will be refereed to as the piecewise constant approximation method, or the PCA method.

All of the computational examples have the same form for the initial condition. The initial condition assumes a source-free equilibrium where $\vec{x}(0)$ equals:

$$\begin{pmatrix} 1 \\ \frac{\beta_1}{\lambda_1 \Lambda} \\ \frac{\beta_2}{\lambda_2 \Lambda} \\ \vdots \\ \frac{\beta_m}{\lambda_m \Lambda} \end{pmatrix}.$$

The first example models a step-reactivity insertion. In this case, $\rho(t) = \rho_0$ for $t \geq 0$. As the reactivity and source functions are constant, the IVP (2.4) is solved exactly, using the PCA method. For step-reactivity insertions, the following parameters were used: $\Lambda = 0.00002, \beta = 0.007, \beta_i = (0.000266, 0.001491, 0.001316,$ $0.002849, 0.00896, 0.000182)$, and $\lambda_i = (0.0127, 0.0317, 0.155, 0.311, 1.4, 3.87)$ with $m = 6$ delayed groups. We consider three step insertions, one prompt subcritical $\rho = 0.003$, one prompt critical $\rho = 0.007$, and one prompt supercritical $\rho = 0.008$. The results of the PCA method are presented in Tables (3.1), (3.2), and (3.3) and are compared with those of the SCM method proposed by Chao and Attard and with the exact values [5].

For step-reactivity insertions, notice that the roots of the inhour equations need only be calculated once, at time $t = 0$. There is no need to re-calculate the roots at each time step as X_i and D_i do not change. The roots of the inhour equation for the three step reactivity examples are given in Table (3.4). Also, while the SCM method yields an accurate solution, the PCA method yields exact results for any time interval in the case of a step-reactivity insertion.

Consider now the case of a ramp reactivity. Ramp reactivities usually take the form $\rho(t) = \rho_0 t$ for some constant $\rho_0 = \frac{\rho}{\beta}$ which is a given reactivity expressed in dollars [2]. We will use the same parameters that we used in the step-reactivity example, and compare our results to those of Chao and Attard. The ramp reactivity that we consider is a "moderately fast" ramp of \$.01/sec. The computational results for this case are presented in Table (3.5) along with the exact values obtained from [3]. While the ramp reactivity results are not as accurate as those for the step reactivity example, it will be shown that the error in the PCA method is proportional to h^2.

Finally, to provide an interesting example of the accuracy and flexibility of the PCA method, we will examine the sinusoidal reactivity case considered in [2]. The reactivity function in this case has the form:

$$\rho(t) = \rho_0 \sin \frac{\pi t}{T}$$

where T is a half-period. For this reactivity function, we consider only one delayed neutron group, instead of the six as in the previous examples. The parameters for this example are: $\rho_0 = .005333$ (or equivalently, 0.68 dollars), $\beta = 0.0079$, $\lambda = 0.077$, $\Lambda = 10^{-3}$ and $T = 50$. The initial conditions for the computation were selected to be at equilibrium as previously described. The computation was run for a period of 100 seconds, and Figure 3.1 plots the neutron density with respect to time. Figure (3.1) agrees with the figure provided in [2]. In Figure 3.2, the precursor density for this problem is plotted along with the corresponding neutron density. These results were obtained for a time step $h = 1$ sec. No significant changes in the figures occur if a smaller time step is used.

Table 3.1: A comparison using a prompt subcritical step reactivity, $\rho = 0.003$

$\rho = 0.003$			
Time (s)	SCM	PCA	Exact
$t = 1$	2.2254	2.2098	2.2098
$t = 10$	8.0324	8.0192	8.0192
$t = 20$	2.8351×10^1	2.8297×10^1	2.8297×10^1

Table 3.2: A comparison using prompt critical step reactivity, $\rho = 0.007$

$\rho = 0.007$			
Time (s)	SCM	PCA	Exact
$t = 0.01$	4.5001	4.5088	4.5088
$t = 0.5$	5.3530×10^3	5.3459×10^3	5.3459×10^3
$t = 2$	2.0627×10^{11}	2.0591×10^{11}	2.0591×10^{11}

Table 3.3: A comparison using a prompt supercritical step reactivity,$\rho = 0.008$

$\rho = 0.008$			
Time (s)	SCM	PCA	Exact
$t = 0.01$	6.2046	6.0229	6.0229
$t = 0.5$	1.4089×10^3	1.4104×10^3	1.4104×10^3
$t = 2$	6.1574×10^{23}	6.1634×10^{23}	6.1634×10^{23}

Table 3.4: Solutions to the inhour equation for several step reactivities

$\rho = 0.003$	$\rho = 0.007$	$\rho = 0.008$
-0.13513	-0.01312	-0.01307
-0.49175	-0.03926	-0.03827
0.12353	-0.14053	-0.13723
-0.16620	-0.76952	-0.65721
-1.14757	-3.17734	-2.54004
-3.72268	11.6442	-5.15004
-200.7647	-13.2449	52.80352

Table 3.5: A comparison using a moderately fast ramp reactivity

t (s)	θ Weighting $h = 0.01$	θ Weighting $h = 0.1$	PCA $h = 0.01$	PCA $h = 0.1$	Exact
2	1.3382	1.3383	1.3382	1.3305	1.3379
4	2.2286	2.2290	2.2278	2.2117	2.2283
6	5.5830	5.5885	5.5802	5.5230	5.5815
8	4.2811×10^1	4.3215×10^1	4.2772×10^1	4.2049×10^1	4.2781×10^1
9	4.8816×10^2	5.0636×10^2	4.8735×10^2	4.7639×10^2	4.8745×10^2

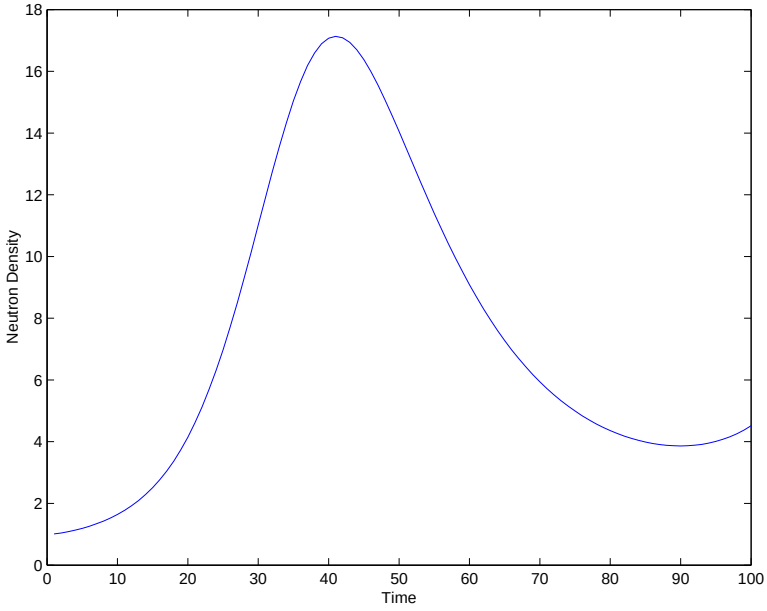

Figure 3.1: Neutron densities based on a sinusodial reactivity

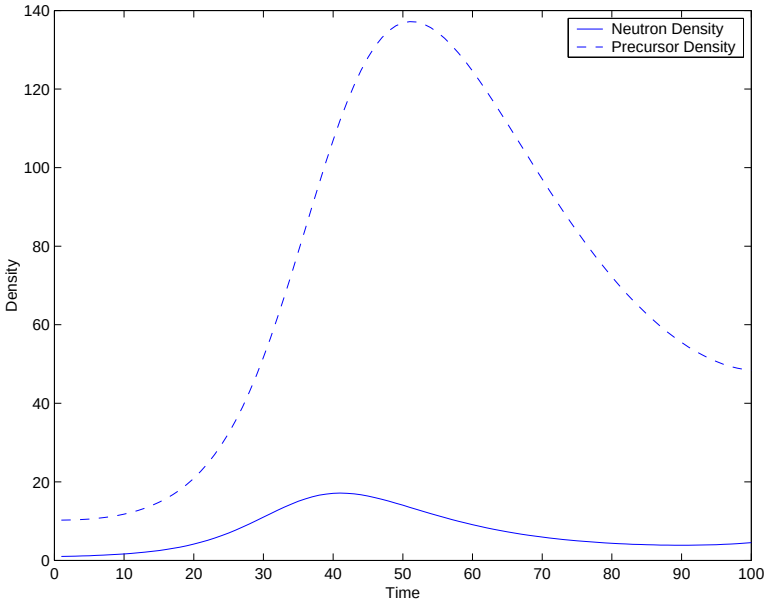

Figure 3.2: Precursor density together with neutron density

CHAPTER IV
ERROR ANALYSIS

Consider the matrix formulation of the point kinetics equations

$$\frac{d\vec{x}}{dt} = A\vec{x} + B(t)\vec{x} + \vec{F}(t) \tag{4.1}$$

$$\vec{x}(0) = \vec{x}_0.$$

In addition, consider the matrix form of the PCA scheme:

$$\frac{d\hat{\vec{x}}}{dt} = A\hat{\vec{x}} + B_i\hat{\vec{x}} + \vec{F}_i \tag{4.2}$$

$$\hat{\vec{x}}(t_i) = \hat{\vec{x}}_i.$$

for $t_0 = 0, t_1 = h, t_2 = 2h, \ldots$ and $h = t_{i+1} - t_i$. Subtracting equations (4.1) and (4.2) yields:

$$\frac{d\vec{\varepsilon}}{dt} = A\vec{\varepsilon} + (B(t) - B_i)\vec{x} + B_i\vec{\varepsilon} + \vec{F}(t) - \vec{F}_i \tag{4.3}$$

$$\vec{\varepsilon}(t_i) = \vec{\varepsilon}_i$$

where $\vec{\varepsilon}(t) = \vec{x}(t) - \hat{\vec{x}}(t)$, $\vec{\varepsilon}(0) = \vec{\varepsilon}_0 = \vec{0}$ and $\vec{x}(t)$ is the exact solution.
To simplify the notation and calculations, let $\vec{g}_i(t) = (B(t) - B_i)\vec{x}(t) + \vec{F}(t) - \vec{F}_i$. This change yields the system:

$$\frac{d\vec{\varepsilon}}{dt} = (A + B_i)\vec{\varepsilon} + \vec{g}_i(t) \tag{4.4}$$

$$\vec{\varepsilon}(t_i) = \vec{\varepsilon}_i$$

Suppose that $B_i = B(t_{i+\frac{1}{2}})$, $F_i = \vec{F}(t_{i+\frac{1}{2}})$ where $t_{i+\frac{1}{2}} = \frac{t_i + t_{i+1}}{2}$. Then, by our piecewise constant approximation, we see that $\vec{g}_i(t_{i+\frac{1}{2}}) = \vec{0}$.

Now consider equation (4.4). Multiplying by the appropriate integrating factor yields:

$$\frac{d}{dt}\left[e^{-(A+B_i)t}\vec{\varepsilon}\right] = e^{-(A+B_i)t}\vec{g}_i(t).$$

Integrating from t_i to t_{i+1} gives us:

$$e^{-(A+B_i)t_{i+1}}\vec{\varepsilon}_{i+1} - e^{-(A+B_i)t_i}\vec{\varepsilon}_i = \int_{t_i}^{t_{i+1}} e^{-(A+B_i)t}\vec{g}_i(t) \quad dt$$

By adding $e^{-(A+B_i)t_i}\vec{\varepsilon}_i$ to both sides and dividing through by $e^{-(A+B_i)t_{i+1}}$, we obtain the form:

$$\vec{\varepsilon}_{i+1} = e^{-(A+B_i)h}\vec{\varepsilon}_i + \int_{t_i}^{t_{i+1}} e^{-(A+B_i)(t_{i+1}-t)}\vec{g}_i(t) \quad dt$$

Now let $\vec{y}_i(t) = e^{-(A+B_i)(t_{i+1}-t)}\vec{g}_i(t)$ be the vector of length $m+1$ and let $y_{i,j}(t)$ be the j^{th} component of $\vec{y}_i(t)$ for $j = 1, 2, \ldots, m+1$. Notice that $y_{i,j}(t_{i+\frac{1}{2}}) = 0$ for $j = 1, 2, \ldots, m+1$ as $\vec{g}_i(t_{i+\frac{1}{2}}) = 0$. Now consider using the midpoint rule to approximate the integral. We obtain that,

$$\int_{t_i}^{t_{i+1}} y_{i,j}(t)dt = hy_{i,j}(t_{i+\frac{1}{2}}) + \frac{h^3}{24}\frac{d^2}{dt^2}\left[y_{i,j}(\xi_{i,j})\right] = \frac{h^3}{24}\frac{d^2}{dt^2}\left[y_{i,j}(\xi_{i,j})\right]$$

.

where $\xi_{i,j} \in [t_i, t_{i+1}]$. This implies that

$$(\vec{\varepsilon}_{i+1})_j = (e^{-(A+B_i)h}\vec{\varepsilon}_i)_j + \frac{h^3}{24}\frac{d^2}{dt^2}\left[y_{i,j}(\xi_{i,j})\right] \quad \text{for} \quad j = 1, 2, \ldots, m+1. \tag{4.5}$$

Let

$$\tau = \max_{1\le j\le m+1} \max_{0\le t,t_i\le t_f} \frac{1}{24}|y_{i,j}(t)|.$$

Then,

$$\|\vec{\varepsilon}_{i+1}\|_\infty \le \|e^{-(A+B_i)h}\vec{\varepsilon}_i\|_\infty + h^3\tau$$

and we can rewrite equation(4.5) in the following form:

$$\|\vec{\varepsilon}_{i+1}\|_\infty \le \|e^{-(A+B_i)h}\|_\infty\|\vec{\varepsilon}_i\|_\infty + h^3\tau.$$

Then, because $\|e^{At}\| \le e^{\|A\|t}$ for $t \le 0$ we obtain

$$\|\vec{\varepsilon}_{i+1}\|_\infty \leq e^{h\|A+B_i\|_\infty}\|\vec{\varepsilon}_i\|_\infty + h^3\tau.$$

Now if we let $k = \max_{0\leq t\leq t_f} \|A + B(t)\|$ we obtain:

$$\|\vec{\varepsilon}_{i+1}\|_\infty \leq e^{kh}\|\vec{\varepsilon}_i\|_\infty + h^3\tau. \tag{4.6}$$

$$\|\vec{\varepsilon}_0\| = 0$$

We can equivalently obtain:

$$\|\vec{\varepsilon}_{i+1}\|_\infty \leq h^3\tau\left(\frac{e^{kh(i+1)}-1}{e^{kh}-1}\right)$$

giving us:

$$\|\vec{\varepsilon}_{i+1}\|_\infty \leq \frac{h^2\tau}{k}(e^{kt_f}-1).$$

Thus, the error in the PCA method is proportional to h^2 as previously stated.

We can most clearly test the error estimate by employing the PCA method to solve a simple problem. Consider, the PCA method applied to the following IVP:

$$\frac{d\vec{y}}{dt} = A(t)\vec{y}$$
$$\vec{y}(0) = [1,1]^T$$

where

$$A(t) = \begin{bmatrix} t-1 & 1 \\ 1 & -1 \end{bmatrix}.$$

This IVP corresponds to the one delayed group, ramp input reactivity with the following parameters: $\beta = 1 \quad \Lambda = 1$ and $\lambda_1 = 1$. The reactivity function is $\rho(t) = t$. The computational results obtained using the PCA method are given in Table (4.6). Clearly, as h decreases by a factor of two, the error in the PCA approximation decreases by a factor of four. This simple example supports the results of the formal error analysis.

Table 4.1: Order h^2 computational results

h	PCA Approximation	Error
0.1	4.52525	0.000166
0.05	4.52749	0.000042
0.025	4.52788	0.000010

CHAPTER V
SUMMARY AND CONCLUSION

The PCA method is clearly an effective numerical method for solving the point kinetics equations. The method is efficient, simple, and accurate. The PCA Method yields exact results in step reactivity insertions and the method has accuracy of order h^2 for problems with time-varying reactivities.

In possible future work, the reactivity and source functions could perhaps be approximated by piecewise linear functions. The implementation of such a method may be more complicated, but exact results could be obtained in the ramp reactivity case along with the step reactivity case.

BIBLIOGRAPHY

[1] Ahmed Ebrahim Aboanber and Abdallah Alsayed Nahla, "Generalization of the analytical inversion method for the solution of the point kinetics equations," *Journal of Physics A: Mathematical and General* **35** (2002), 3245-3263.

[2] D.L. Hetrick, *Dynamics of Nuclear Reactors*, The University of Chicago Press, Chicago (1971)

[3] J. Sánchez, "On the Numerical Solution of the Point Kinetics Equations by Generalized Runge-Kutta Methods," *Nucleaer Science and Engineering* **103** (1989), 94-99.

[4] J.A.W. da Nórbrega, "A New Solution of the Point Kinetics Equations," *Nuclear Science and Engineering* **46** (1971), 366-375.

[5] Yung-An Chao and Anthony Attard, "A Resolution to the Stiffness Problem of Reactor Kinetics," *Nucleaer Science and Engineering* **90** (1985), 40-46.

```
function y = piecewise_const(lambda, beta,beta_sum, L, target, h,
rho_case, f_case, init_cond,rval)

%% This function will calculate the solution to the point kinetics equation
%% starting at time = 0.  The following is a summary of the arguments for the
%% function:
%%  lambda = a vector of the decay constants for the delayed neutrons
%%  beta   = a vector containing the delayed-neutron fraction
%%  beta_sum = the sum of all of the betas
%%  L      = neutron generation time
%%  target = the final, target time of the function
%%  h      = step size
%%  rho_case = this value will determine the type of time dependent
%%         reactivity for the program.  A full listing of these
%%                   various values can be found in rho.m
%%  f_case  = similar to "rho_case"  but it determines the source term
%%                to be used


%% 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; time = 0;
f_hat = zeros(m,1);
d_hat= zeros(m,m);
big_d = zeros(m,m);
i = 1;
iterations = target / h

result = zeros(m+1,iterations);
```

```
%% Begin time dependent iterations
d=rval;

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

    f_hat(1) = source;
    big_d_inv = zeros(m,m);

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

    %% Compute next time step
```

```
    x = (Y * d_hat * Y_inv)*(x + (Y*big_d_inv*Y_inv*f_hat)) -
    (Y*big_d_inv*Y_inv*f_hat);

    %% Store results in a matrix
    for j = 1:m
    result(1,i)  = time;
    result(j+1,i) = x(j);
    end

    %%Update counters
    i = i + 1;
    end
    y=result;
-------------------------------------------------------------------------
function y = ev2(lambda,L,  beta, evals)
%%This is a simple function that calculates the eigenvectors using the
%%appropriate forms.

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

y = coeff;
-------------------------------------------------------------------------
function y = f(case_number, time)
%% This is the source function for our solution. It works just like rho.m
%% in that the case_number determines the function to use.

   %% case_number=1   :  f = 0

if case_number == 1
   result = 0;
end

y=result;
-------------------------------------------------------------------------
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 derivitive
%% 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 mehtod
%%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 respectivly

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.  If an
%%  unknown case is encountered the function returns 0.

%%  case_number = 1 :  Step reactivity of rho = 0.003
%%  case_number = 2 :  Step reactivity of rho = 0.0055
%%  case_number = 3 :  Ramp reactivity
```

```
if case_number == 1
   result = .003;
end
if case_number == 2
   result = .0055;
end
if case_number == 3
   result = 0.1*beta_sum*t;
end
y=result;
```

```
function y = rootfinder(coeff, init, tol)
%% This is a simple wrapper function that takes an coefficent 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);

--------------------------------------------------------------------------
%% pcrun.m
%% This is a dummy script to run the piecewise constant function
```

```
%% ---------------------- Chao and Attard---------------------------------------
%%lambda = [0.0127, 0.0317, 0.115, 0.311, 1.4, 3.87]; L = 0.00002;
%%beta = [0.000266, 0.001491, 0.001316, 0.002849, 0.000896,
%%0.000182];
%%beta_sum = 0.007;

%% ----------------------  Norbrega Thermal Reactor-------------------------
%%lambda = [0.0127, 0.0317, 0.115, 0.311, 1.40, 3.87];
%%L = 5e-4;
%%beta = [2.850e-4, 1.5975e-3, 1.410e-3, 3.0525e-3, 9.600e-4, 1.950e-4];
%%beta_sum = 0.00750;

%%----------------------  Norbreaga Fast Reactor----------------------------
%%lambda = [0.0129, 0.0311, 0.134, 0.331, 1.26, 3.21];
%%L = 1e-7;
%%beta = [1.672e-4, 1.232e-3, 9.504e-4, 1.443e-3, 5.434e-4, 1.540e-4];
%%beta_sum = 0.00440;

%%------------------------Two Group-----------------------------------
%%lambda = [.077];
%%L = [.0001];
%%beta = [.0079];
%%beta_sum = 0.0079

%% A short code for computing the initial equilibrium conditions

m = length(lambda);
init_cond =  zeros(m+1,1);
init_cond(1) = 1;
for i = 1:m
init_cond(i+1) = beta(i) / (L* lambda(i));
end
```

```
rval = transpose([0,0,0,0,0,0,0]);

z = piecewise_const(lambda, beta, beta_sum L, 1, .1 , 1  , 1
,init_cond, rval)
```