The Dissertation Committee for Stephen Christian Wilson Certifies that this is the approved version of the following dissertation:

Development and Implementation of a Finite Element Solution of the Coupled Neutron Transport and Thermoelastic Equations Governing the Behavior of Small Nuclear Assemblies

Committee:

Steven Biegalski, Supervisor

Eric Becker

Ofodike Ezekoye

Mark Mear

Sheldon Landsberger

Development and Implementation of a Finite Element Solution of the Coupled Neutron Transport and Thermoelastic Equations Governing the Behavior of Small Nuclear Assemblies

by

Stephen Christian Wilson, B.S.; M.S.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2006

Dedication

To my fiancée Kathleen, and my parents Steve and Laura.

Acknowledgements

Thanks to Richard Coats, Sharon Walker and James Dahl of Sandia National Laboratories for providing the impetus and support for this research. Thanks also to my graduate supervisor, Steven Biegalski, and program head, Sheldon Landsberger, for their support and guidance throughout my graduate career.

Development and Implementation of a Finite Element Solution of the Coupled Neutron Transport and Thermoelastic Equations Governing the Behavior of Small Nuclear Assemblies

Publication No._____

Stephen Christian Wilson, PhD.

The University of Texas at Austin, 2006

Supervisor: Steven R. Biegalski

Small, highly enriched reactors designed for weapons effects simulations undergo extreme thermal transients during pulsed operations. The primary shutdown mechanism of these reactors -- thermal expansion of fuel material -- experiences an inertial delay resulting in a different value for the fuel temperature coefficient of reactivity during pulse operation as compared to the value appropriate for steady-state operation. The value appropriate for pulsed operation may further vary as a function of initial reactivity addition. Here we design and implement a finite element numerical method to predict the pulse operation behavior of Sandia Pulsed Reactor (SPR) II, SPR III, and a hypothetical spherical assembly with identical fuel properties without using operationally observed data in our model. These numerical results are compared to available SPR II and SPR III operational data. The numerical methods employed herein may be modified and expanded in functionality to provide both accurate characterization of the behavior of fast

burst reactors of any common geometry or isotropic fuel material in the design phase, as well as a computational tool for general coupled thermomechanical-neutronics behavior in the solid state for any reactor type.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Problem Statement

In many reactor physics applications, the fuel temperature coefficient of reactivity may be characterized by taking into consideration Doppler broadening, neutron spectrum hardening, and simple non-transient thermal expansion effects (Lamarsh 2001).

A major exception to this rule exists in all-metal reactors whose primary mode of operation is pulse (or burst) mode. These reactors, designed to experimentally simulate the various effects of weapons detonations on components and other objects, experience a relatively large thermal energy deposition on a time scale of only tens of microseconds. In such a case, non-transient thermal expansion does not suffice to describe the inertial delay in fuel expansion. Solving the appropriate thermoelastic equations is required to accurately find the fuel displacement as a function of time.

One approach to solving the thermoelastic equations is to utilize a simplified kinetics model and an experimentally determined fuel temperature coefficient of reactivity to provide a power burst function. The power function then directly yields a flat temperature rise as a function of time, which can be inserted into a finite element code to find the displacement in time. While this approach is adequate to solve for stresses and displacements after the reactor has been operated a sufficient number of times to arrive at an operational fuel temperature coefficient of reactivity, it is less valuable in attempting to solve for the coefficient in the design phase prior to operation.

To do so, some version of the neutron transport and thermoelastic equations must be solved simultaneously on a complicated domain to numerically simulate a pulse in its entirety. Attempting to do so in three spatial dimensions and time represents a significant computational and programming challenge. Employing either a one or two-dimensional

solution where applicable may spare significant programming effort and computation time.

1.2 LITERATURE REVIEW

The volume of work available on the dynamic behavior of extreme thermomechanical shock in fast burst reactors has been somewhat limited by the narrow range of the field. The work that has been accomplished regarding this special class of research reactor may be divided into two areas.

First, work done in the neutron kinetics of small, fast burst reactors may be attributed in large part to scientists at Los Alamos and Sandia National Laboratories during the 1960's and 1970's. The equations governing power, energy and temperature are well characterized and have been for decades (Hansen, 1952); (Wimmet, 1960); (Hetrick, 1971). They consist largely of simplified versions of the point-kinetics equations, since such small reactors are especially well suited to the point-kinetics assumption.

The second area of research has been more extensive. A number of studies of the thermomechanical effects of rapid power transients in small reactors, especially the Sandia Pulsed Reactors II and III (SPR II and III) and Los Alamos' Lady Godiva spherical assemblies have been made (Hansen, 1952); (Reuscher, 1969, 1972); (Wimmet, 1992); (Miller, 1994). All of these analyses, whether numerical or analytical, utilized a burst function with some experimentally determined parameters (either fuel temperature coefficient of reactivity or initial reciprocal burst positive period). The analytical work done by Wimmet (1992) further required an observed vibration frequency.

While these methods of finding stresses and displacements in fast burst research reactors are perfectly legitimate, they are less useful as design tools for new reactors or assemblies of different geometry or fuel material. For use in pre-operation design under more stringent modern safety standards, one must necessarily produce a fuel temperature coefficient of reactivity from scratch, in a simultaneous solution of the neutron kinetics and thermoelastic equations.

Supporting work done in the fields of uranium alloy metallurgy and safety analysis of fast burst research reactors has been extensive, and is repeatedly referenced in this report.

1.3 STATEMENT OF WORK

This work seeks to develop a set of integrated, time-dependent finite element codes to simultaneously solve the equations governing the behavior of small, highly enriched spherical and cylindrical nuclear assemblies. The goal is to reproduce known reactor behaviors to a reasonable degree of accuracy, and to lay the groundwork for more sophisticated solutions taking advantage of the program logic developed herein.

The reactors for which operational data is most readily available belong to the SPR reactor series operated at Sandia National Laboratories (SNL). Numerical simulations of these reactor behaviors will be most valuable in determining the accuracy of the code. Since data and analytical solutions are available for spherical arrays, they will provide a secondary check of the validity of the algorithm employed in the finite element codes.

Chapter 2: SPR Reactors

2.1 HISTORY

Shortly after the advent of nuclear weapons, the desire to simulate their radiation side-effects led to the construction of a unique group of reactors designed specifically to do so: fast burst reactors. Instead of large, water-moderated systems designed for power production, these reactors are small, unmoderated, and consist of highly enriched metal fuel and metal (stainless steel or aluminum) support structures. Their primary mode of operation is burst mode, wherein the reactor is moved from a subcritical state to prompt supercritical via a large addition of reactivity, usually in the form of a burst rod or reflector. The ensuing prompt supercritical excursion produces a neutron fluence in an experimental cavity on an order approaching that of a weapon and closely mirroring its neutron energy spectrum. These conditions simulate the effects of a weapon on small components and objects.

Unlike virtually all other nuclear reactors, the primary and essential shutdown mechanism in a fast burst reactor is thermal expansion of fuel. While control elements and other mechanisms for a safe approach to steady-state criticality do exist in these reactors, the only shutdown characteristic capable of responding fast enough to the rapid power increase inherent to burst operation is thermal expansion. Thermal expansion increases the volume and decreases the density of the system, which in turn increases the neutron leakage and shuts down the reaction.

Sandia National Laboratory has operated a series of fast burst reactors over the past four decades. Each reactor is prefixed SPR, and three have operated since the inception of the SNL burst reactor program.

2.1.1 SPR

The first fast burst reactor operated at SNL, simply known as the SPR, was also referred to as Godiva III due to its similarity to Los Alamos National Laboratory's Godiva II reactor. The SPR was operated at SNL from 1961 to 1967. The fuel was highly enriched (93.2% ^{235}U) cast uranium, with a total mass of 57.2 kg.

The reactor geometry was a right circular cylinder of radius 8.9 cm and height 14.3 cm. Three uranium bolts fastened the assembly together, and to a steel support plate. Four cylindrical cavities in the bottom accommodated the safety block, two control rods, and a burst rod. Figure 2.1 provides a rough illustration of the reactor configuration.

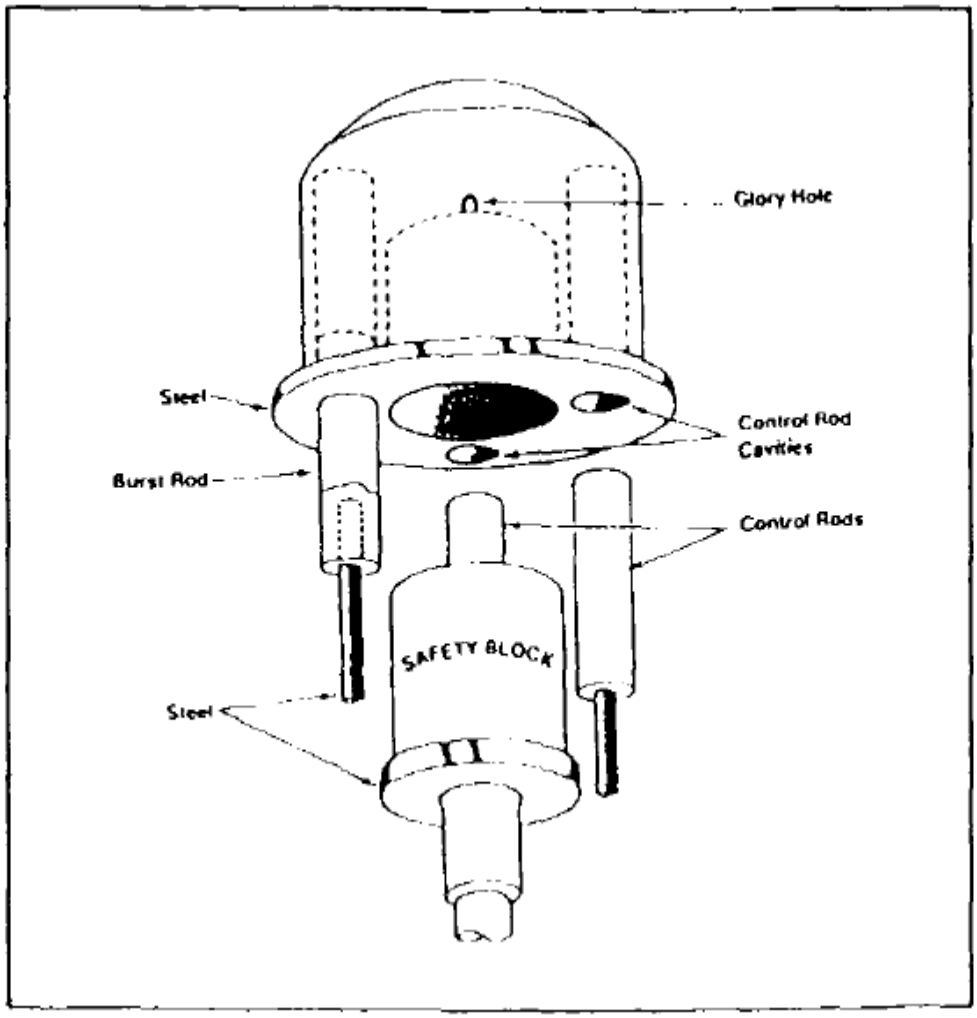


Figure 2.1: Diagram of SPR fuel assembly (Reuscher and Schmidt, 1994)

Compared to Godiva II, SPR fuel components were machined to reduce core reactivity and increase control rod worth from \$2 to \$3 (see Chapter 3, Section 3.1.3 of this dissertation for definition of reactivity terms). The reactor stand was mounted on a hydraulic elevator capable of lowering the entire reactor into a shielded concrete pit immediately after operation. A through-hole, approximately 0.8 cm^2, allowed irradiation of small objects. The fuel was cadmium plated to reduce oxidation and decouple the system from room return neutrons (cadmium is an effective neutron absorber). The

design burst for SPR was 2 x 10^{16} fissions, which produced a temperature rise of 110 K in the fuel mass (Reuscher and Schmidt, 1994).

2.1.2 SPR II

The second fast burst reactor operated by SNL was Sandia Pulsed Reactor II, or SPR II. It was designed and constructed by SNL, and became operational in 1967. The SPR-II reactor core is solid metal fuel enriched to 93% ^{235}U. The uranium is alloyed with ten percent by weight molybdenum in a process designed to freeze the alloy in uranium's body centered cubic gamma phase.

This gamma-phase stabilization provides superior material strength in terms of ultimate yield. More importantly, it eliminates the anisotropic expansion behavior of uranium's room-temperature tetragonal alpha phase. This greatly reduces the negative effects of thermal cycling, thereby increasing the effective lifetime of the fuel.

The SPR II core consists of six fuel plates, divided into two assemblies of three plates each. The three lower plates are attached to an electromechanical drive mechanism designed to allow the entire lower block, known as the safety block, to break away during pulse operations with temperature changes exceeding 150 K.

Five vertical holes through the core assembly accommodate the irradiation cavity, three control rods, and one burst rod. The burst rod has its own hydraulic drive to allow for high reactivity addition rates. As stated above, the primary shutdown mechanism of this reactor is the negative fuel temperature coefficient of reactivity inherent to its fuel.

During operation of SPR II, a ^{10}B loaded silastic material shrouds the reactor. This shroud provides channels for room-temperature nitrogen gas functioning as coolant, and also serves to decouple the reactor core from room return neutrons. The reactor's central cavity radius is 1.9 cm, and the core consists of 105 kg of U-10 Mo fuel alloy (Reuscher and Schmidt, 1994). Figure 2.2 illustrates the reactor's configuration in cross-section.

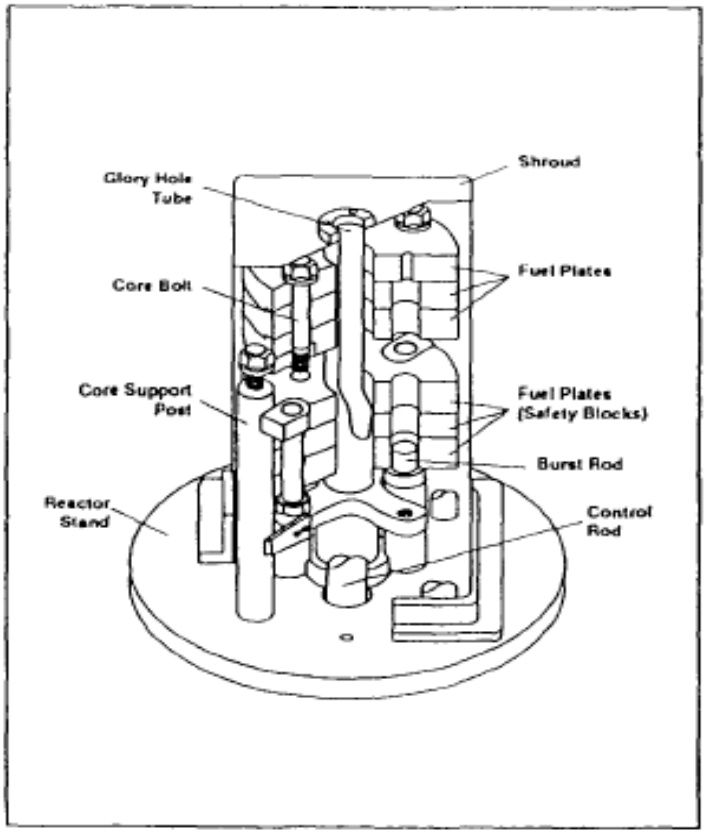


Figure 2.2: Cutaway of SPR II reactor with shroud (Reuscher and Schmidt, 1994)

Operators pulsed SPR II more than five thousand times over the course of its lifetime. The measured operational data gathered from this extensive history provided design insights for future reactors. Specifically, the design fluence of 10^{15} n/cm^2 resulted

in temperature rises of 560 K above reference. These severe thermal transients resulted in cracked fuel plates and rods. Cracks tended to propagate from the interior surfaces outward. The existence of these cracks led to the thermomechanical analyses performed on SPR II, which not only attempted to characterize the internal stresses and displacements, but also to predict fuel failure modes with some degree of certainty (Burgreen, 1962); (Reuscher, 1969, 1972); (Wimmet 1992). The analyses performed prior to the next SPR reactor's design provided insight on the minimization of the negative consequences of high-yield bursting in SPR II.

The shutdown characteristics of SPR II can be split into two categories: steady-state and "inertial". The steady-state fuel temperature coefficient of reactivity is approximately -$0.00293 per K, and the inertial coefficient is approximately -$0.0005 per K. The steady-state fuel temperature coefficient of reactivity holds during relatively long-term, delayed critical operations, where no inertial expansion effects are present, whereas the inertial fuel temperature coefficient of reactivity holds during pulse operations, when inertial effects are significant. The value given here for the inertial fuel temperature coefficient of reactivity represents the lower bound of those encountered during pulse operation.

2.2 SPR III

Sandia's third SPR reactor (a fourth and fifth, SPR-IV and SPR III-M, were designed but never operated) was the product of the design lessons mentioned above, and became operational in 1975. The SPR III core consists of 18 fuel plates mechanically fastened into two halves of nine plates each. The fuel is the same as SPR II fuel: 93% ^{235}U uranium alloyed with ten percent by weight molybdenum. The plates are aluminum ion

coated to reduce oxidation, and the total fuel mass of SPR III is 252 kg. The nine upper plates are stationary, while the nine lower plates (the safety block) are attached to a drive mechanism.

The control elements, instead of rods as in SPR and SPR II, are external copper reflectors. The burst element is an external aluminum reflector with an electromechanical drive to achieve high reactivity addition rates. The use of reflectors instead of in-core rods eliminates the need for extra cavities in the fuel mass, which was a primary contributor to the cracking of SPR II fuel plates.

The primary shutdown mechanism, as in all fast burst reactors, remains the negative fuel temperature coefficient of reactivity inherent to SPR fuel.

SPR III utilizes a shroud for much the same purposes as SPR II: to decouple the reactor from backscattered neutrons and to provide coolant channels. SPR III's shroud is aluminum coated with a ^{10}B loaded silastic material.

One of the significant improvements of SPR III over SPR II is the size of the experimental cavity. With an increase in diameter to 17 cm, the volume of SPR III's "glory hole" is approximately forty-four times that of SPR II. Figure 2.3 shows the reactor configuration in cross-section.

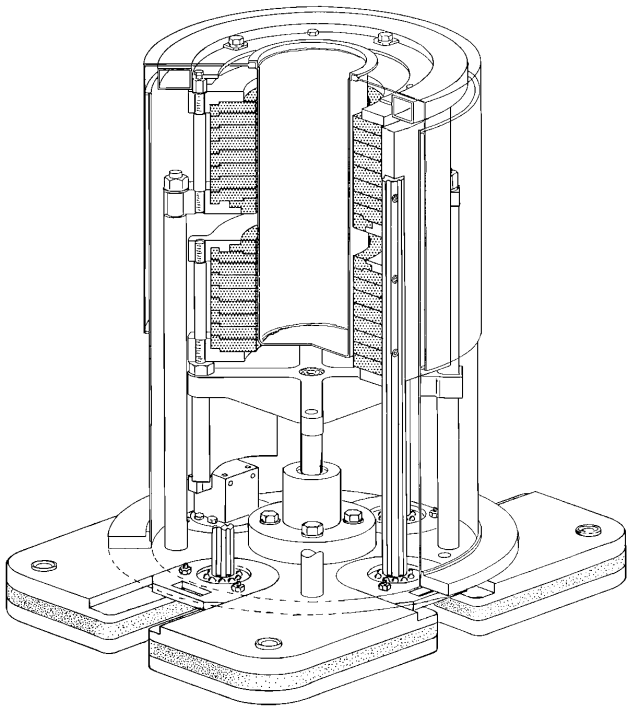


Figure 2.3: SPR III cutaway view (Ford, *et al.* 2003)

Since SPR II and SPR III are the primary subjects of this work, we include in-depth information on the material data required for subsequent solution of the thermoelastic equation, especially that pertaining to the properties of U-10 Mo fuel alloy.

2.2.1 U-10 Mo material characteristics

This particular alloy of uranium was chosen for SPR II and SPR III over others for a number of reasons:

- A large inherent negative temperature coefficient of reactivity sufficient to terminate prompt supercritical pulses within the operational range of the reactor;
- Metallurgical properties that ensure integrity of the material at high temperatures, especially during fast temperature transients experienced in pulsing;
- Minimization of stresses and thermomechanical shock during prompt supercritical pulses;
- Fail safe for accidental prompt excursions;
- Not subject to gross segregation;
- Not subject to ratcheting during thermal cycling;
- Within the scope of existing fuel-fabrication technology.

While alloying with certain other elements (niobium, zirconium and ruthenium in particular) may produce other favorable characteristics, the cost and added complexity made it unfeasible. U-10 Mo was therefore selected as the state-of-the-art fuel for SPR II, and remained the best choice for SPR III (Lundin, 1962); (Carver and Taxelius, 1963); (Dickinson, *et al.* 1967); (Minnema, *et al.* 2001).

The stability of the gamma phase over many pulse operations is particularly important. Fortunately, it has been found that operation in a radiation environment tends to favor gamma phase stability instead of transforming it back to alpha. In fact, operation in a radiation environment has been found to reverse such a transformation (Konobeevsky, *et al.* 1956); (Bleiberg, *et al.* 1956). We can observe in Figure 2.4 the phase diagram of uranium-molybdenum alloys as a function of the atom percent content of molybdenum.

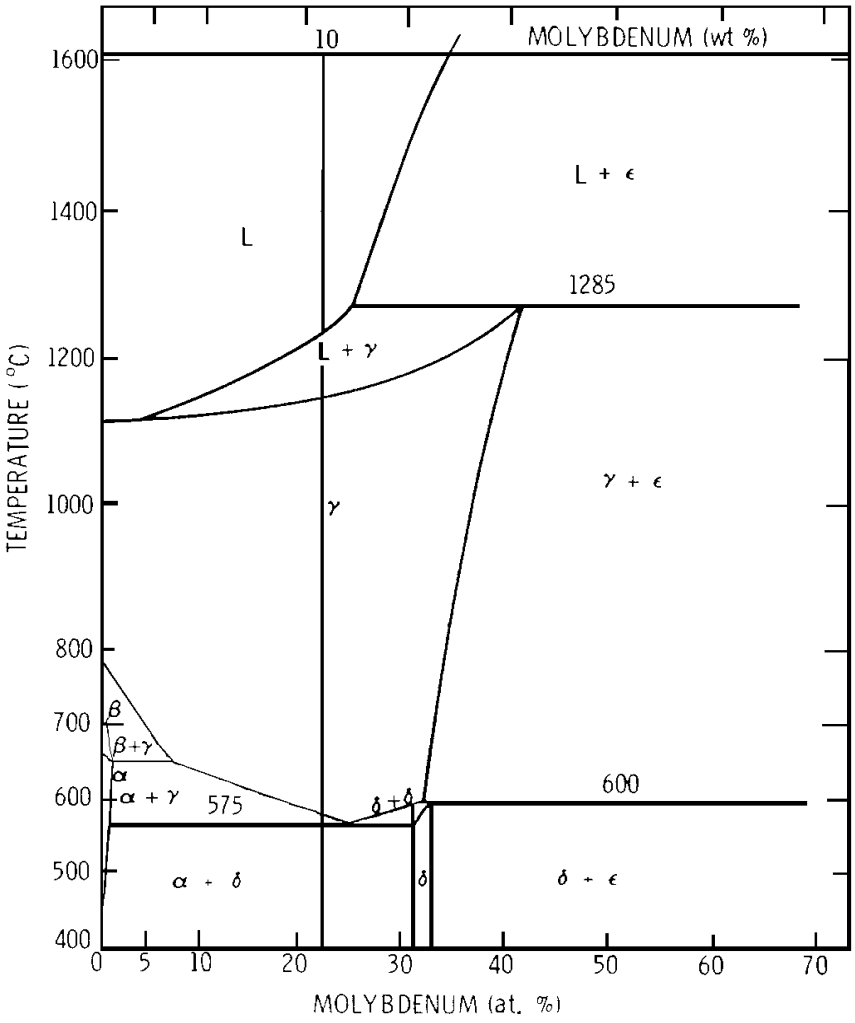


Figure 2.4: Uranium-molybdenum phase diagram (Peterson and Steele, 1963)

Alloying uranium and molybdenum under the appropriate heat treatment maintains the gamma phase at room temperature and the anisotropic thermal expansion inherent to the alpha phase is avoided. U-10 Mo can be stabilized in the gamma phase and under thermal cycling has the dimensional stability characteristic of body-centered cubic structures (Peterson and Steele, 1963); (Lehman, *et al.* 1964).

To understand the stability of the uranium-molybdenum alloy, it is most useful to observe the appropriate time at temperature transformation curves. Even at the 450 °C outer-envelope operating temperature of SPR III, it would take more than nine hours to transform the fuel alloy back to alpha phase (Van Thyne and McPherson, 1957). Figure 2.5 illustrates time-at-temperature transformation curves for various alloys of uranium and molybdenum.

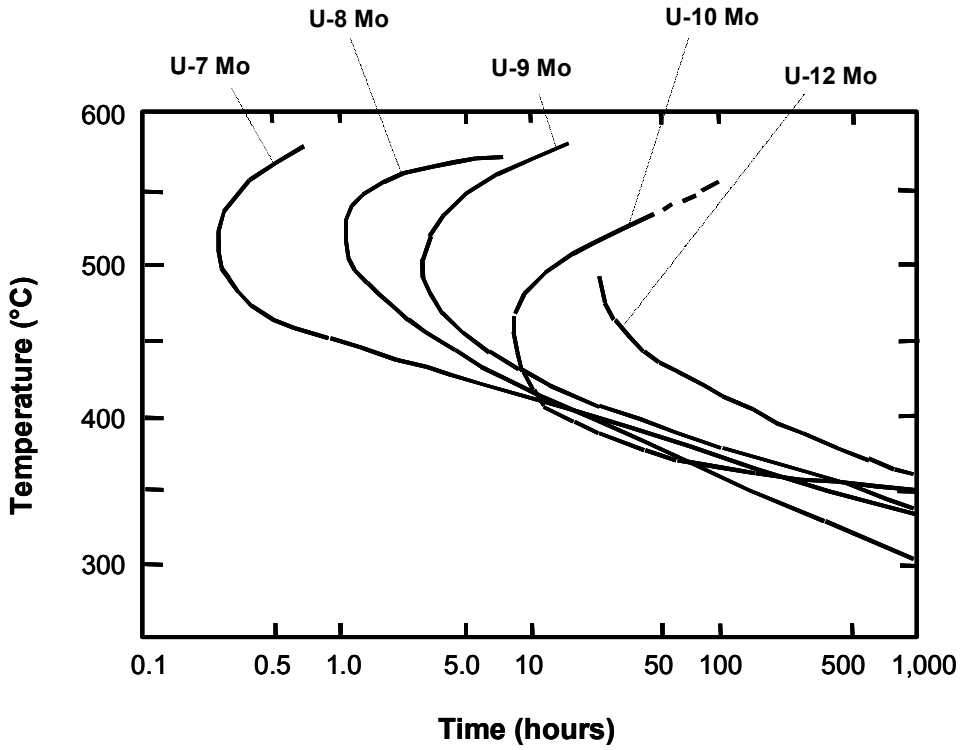


Figure 2.5: Time-at-temperature transform curves for uranium-molybdenum alloys (Van Thyne and McPherson, 1957)

This metallurgical information is significant to our current project because it allows us to assume the material is homogeneous throughout any number of pulse operations. If the material were multiphasic, the solid model would necessarily be more

complex to achieve even an approximation of the exact behavior. Since SPR III would never operate at such a high temperature for more than a few minutes, it is safe to assume the material remains homogeneous not only through a pulse, but also throughout the lifetime of the fuel.

The isotropic expansion properties of U-10 Mo are relatively simple to model, as they remain essentially linear through 600 °C. Therefore, a temperature-dependent coefficient of expansion is not truly required (although one is used in our model). Figure 2.6 illustrates the expansion properties of U-10 Mo.

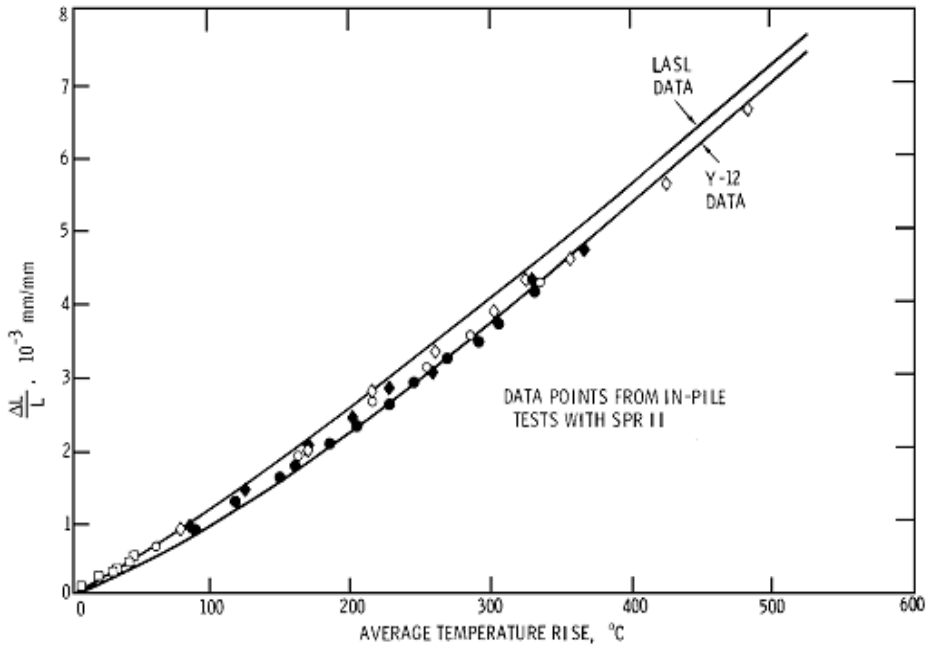


Figure 2.6: Expansion properties of U-10 Mo alloy (Reuscher, 1973)

While the Poisson's ratio of U-10 Mo may be assumed constant at approximately 0.38 (Hoge, 1965), the modulus of elasticity varies significantly with temperature. This variation must be accounted for to reproduce accurate displacement due to temperature

transients in the fuel. Figure 2.7 shows the variation of the elastic modulus of U-10 Mo with temperature.

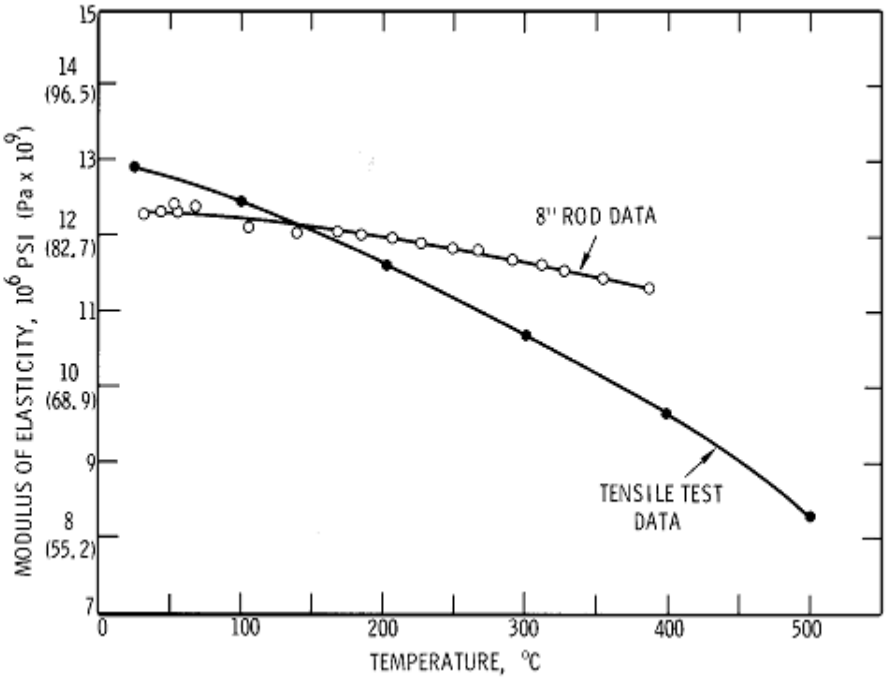


Figure 2.7: Variation of modulus of elasticity of U-10 Mo with temperature (Reuscher, 1973)

The temperature dependence of the specific heat of U-10 Mo is particularly important when applying the neutron kinetics equations. Figure 2.8 illustrates this temperature dependence through the operational temperature range of SPR III.

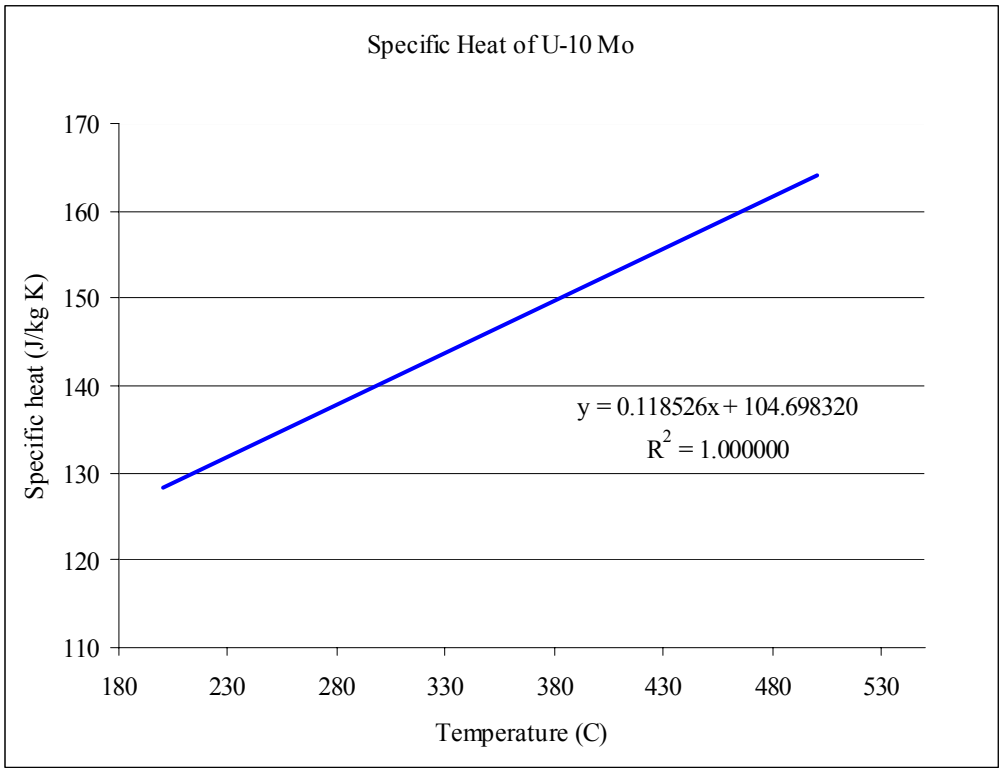


Figure 2.8: Two-point fit of temperature dependence of the specific heat of U-10 Mo alloy (Wilkinson, 1962)

2.2.2 SPR III reactor characteristics

The SPR III fuel assembly forms a hollow right circular cylinder 36.83 cm high, with an inner radius of 8.89 cm and an outer radius of 14.859 cm everywhere except the two A plates, where it is 12.446 cm (see Figures 2.9 and 2.10). The assembly is segmented into 18 separate fuel rings, held in two blocks of nine rings each.

The upper half of the reactor and its nine fuel rings are held stationary by four steel posts fixed to the support assembly. The lower half of the reactor, also called the safety block, is attached to an electromechanical drive unit with a movement range of 8.89 cm. Figure 2.9 illustrates the entire fuel assembly, and Figure 2.10 shows the upper

half of the core. Note in particular the way each successive fuel plate is stacked upon the next. Instead of lying flush, each fuel plate is only in contact with about 25% of the surface area of its neighbors. This axial segmentation helps eliminate axial stress wave propagation through the assembly by creating discontinuities in the fuel continuum. Reducing interface contact between successive plates via the gaps seen in Figure 2.10 further minimizes axial stress wave propagation.

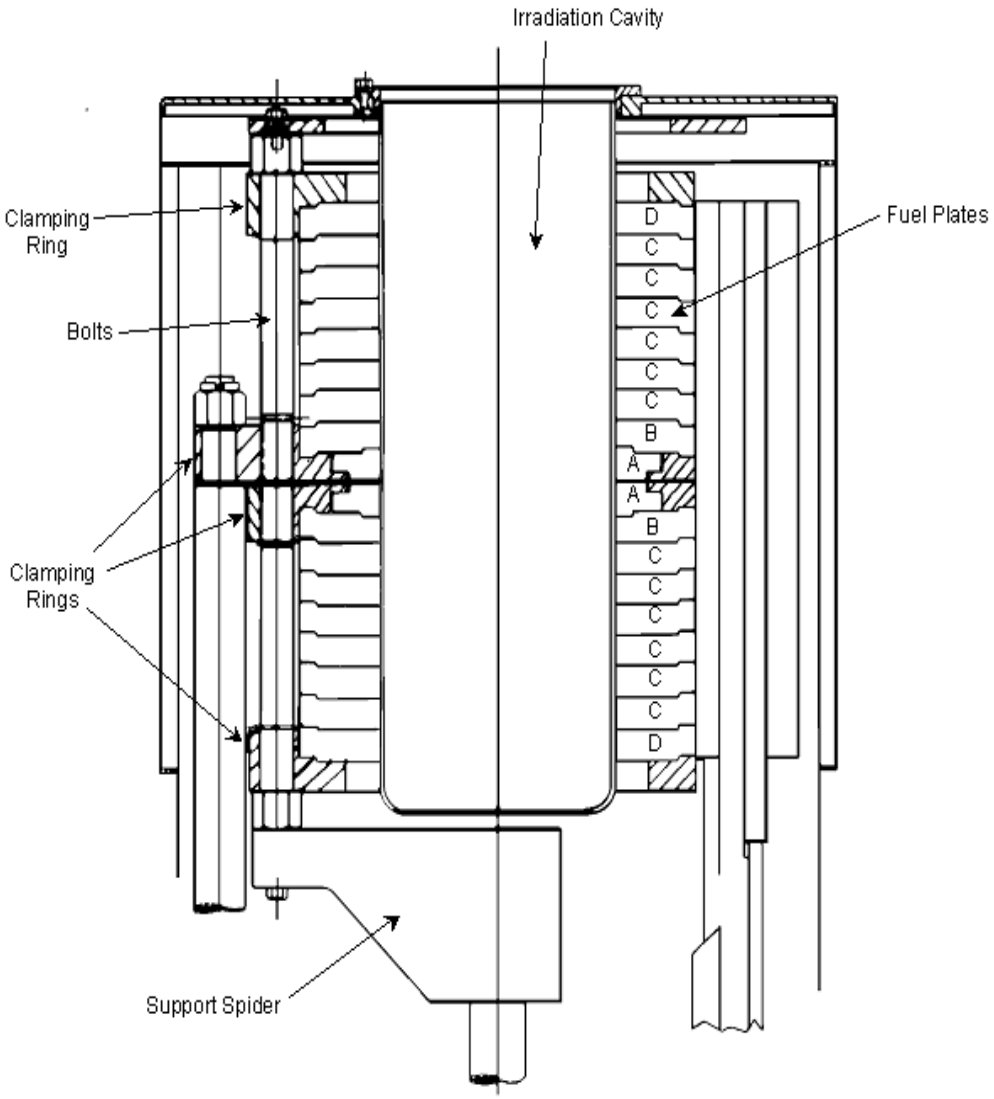


Figure 2.9: SPR III fuel assembly (Ford, *et al.* 2003)

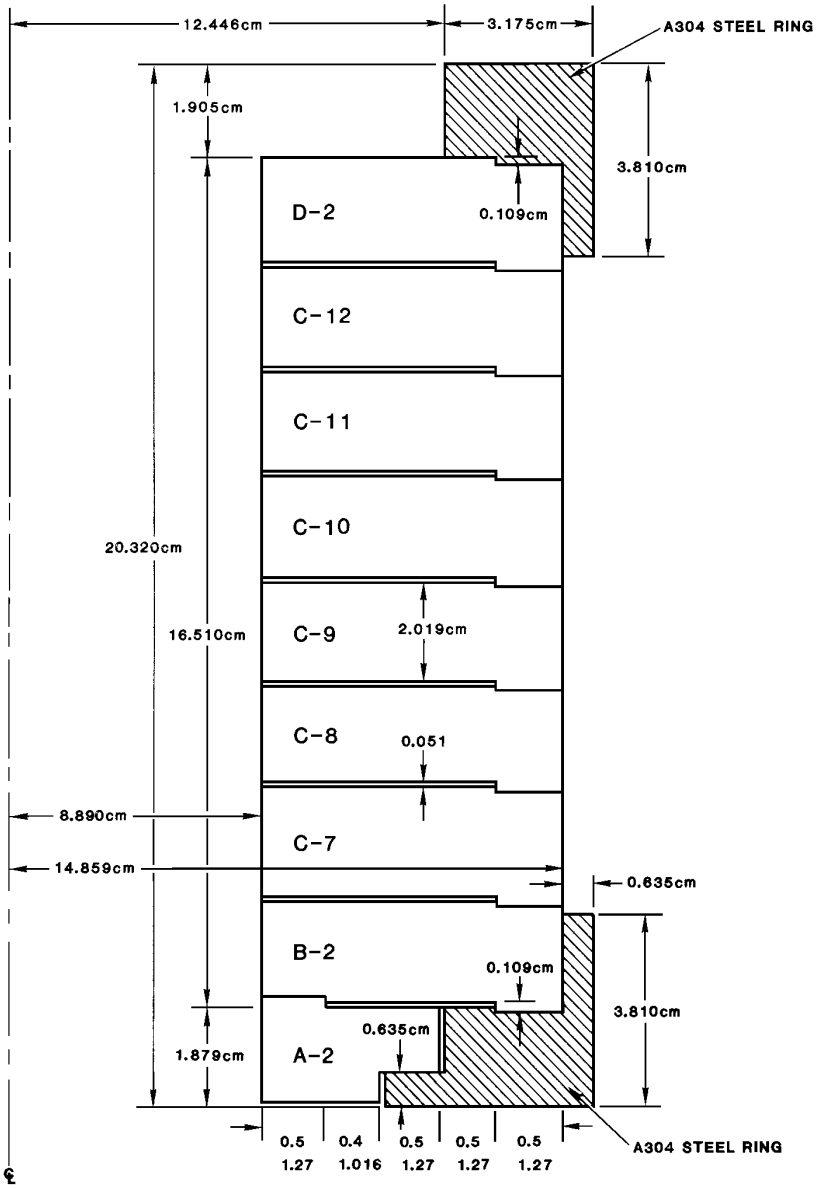


Figure 2.10: Upper half of SPR III fuel assembly (Ford, *et al.* 2003)

Figures 2.9 and 2.10 illustrate the spacing and assembly of the entire SPR III reactor. A zero traction and zero body force in the axial direction may be assumed to solve the appropriate dynamic thermoelastic equation over such a geometry and still give surprisingly accurate results (Wimmet, 1992).

The shutdown characteristics of SPR III can be divided into the same two categories mentioned above for SPR II. The steady-state fuel temperature coefficient of reactivity is -$0.003 per K, and the inertial coefficient is approximately -$0.00044 per K. The inertial coefficient is the result of a fit of operational pulse data to an existing kinetics model.

2.3 SUMMARY

Table 2.1 summarizes the core characteristics of SPR II, SPR III, and the never-produced SPR III-M.

Area	Design Characteristic	Nominal Value		
		SPR II	SPR III	SPR IIIM
Total Core	Height (cm)	20.84	40.64	39.88
	Fuel height (cm)	20.84	36.83	37.34
	Diameter (cm)	20.53	29.72	33.02
	Height/diameter ratio	1.016	1.24	1.13
	Total U-10 Mo mass (kg)	105.54	258.0	294.68
	Dollars subcritical, assembled, rods down (SPR II), control reflectors down (SPR III)	>3.5	~3.5	~3.5
	Delayed neutron fraction	0.0065	0.0065	0.0065
	Prompt neutron lifetime (nanoseconds)	10.4	15	~20
Fuel Ring	**A plate**			
	Thickness (cm)	3.38	1.88	2.74
	Outside diameter (cm)	20.52	24.7	33.02
	Inside diameter (cm)	4.19	17.78	21.34
	U-10 Mo mass (kg)	16.26	6.80	15.50
	B plate			
	Thickness (cm)	3.38	1.88	2.19
	Outside diameter (cm)	20.52	29.72	33.02
	Inside diameter (cm)	4.19	17.78	21.34
	U-10 Mo mass (kg)	16.48	15.0	16.48
	C and D plates			
	Thickness (cm)	3.54	2.03	2.19
	Outside diameter (cm)	20.52	29.72	33.02
	Inside diameter (cm)	4.19	17.78	21.34
	U-10 Mo mass (kg)	16.24	15.3	16.48
Safety Block	Height (cm)	10.42	20.32	19.94
	Fuel height (cm)	10.42	18.41	18.67
	Diameter (cm)	20.52	29.72	33.02
	U-10 Mo mass (kg)	48.98	129.0	147.34
	Shutdown worth ($)	-26.00	-15.00	TBD

Table 2.1: Summary of core design characteristics of SPR reactors (Ford, *et al.* 2003)

Table 2.2 summarizes the relationship between prompt insertion and temperature maximum, and allows us to derive an evolving theoretical shutdown coefficient as a function of insertion size. This data allows us to reproduce observed SPR III behavior

using an appropriate neutron kinetics model, and is the basis for such comparisons throughout this report.

T (C)	SPR II		SPR III	
	Full Width at Half Maximum (FWHM) (μs)	Prompt Reactivity (¢)	FWHM (μs)	Prompt Reactivity (¢)
100	135	4	205	4.5
150	85	5.6	150	6.5
200	65	7	119	7.7
250	53	8	100	8.8
300	48	9	85	9.8
350	45	9.8	81	10.7
400	42	10.5	77	11.6
450	40	11.3	74	12.6
500	38	12.2	71	13.4

Table 2.2: Summary of prompt insertion to maximum temperature relationship for SPR II and SPR III (Ford, *et al.* 2003)

SPR III represents the latest fast burst reactor technology with extensive operational history. Using the data in Table 2.2, we can directly compare any numerical data we produce to a neutron kinetics fit of experimental SPR II and SPR III data. With direct comparison to operational temperature data and fuel displacement data, SPR II and SPR III are the best options for attempting to simulate dynamic fuel expansion behavior during pulse operation of a fast burst reactor.

Chapter 3: Theory

3.1 CONTINUITY EQUATION

The equation governing neutron transport behavior in time over a particular spatial domain derives directly from balance considerations. In its most general conceptual form, it begins as:

> *"Rate of change of neutron density = Neutron production rate – Neutron loss rate"*

Neutron production refers to all sources of neutrons in a reactor: prompt fission neutrons, delayed neutrons produced via the decay of fission product precursors, neutrons returned to the system from reflectors, spontaneous fission neutrons, and any external neutron source, whether artificial or background neutrons from cosmic sources. Neutron loss refers to neutrons leaking from the system, as well as neutrons lost to absorption.

This section contains a brief summary of the equations governing neutron transport, diffusion and point kinetics.

3.1.1 Neutron Transport Equation

The complete neutron transport equation with virtually no limiting assumptions, as derived from control volume balance considerations, is:

$$\frac{1}{v(E)}\frac{\partial\psi(\vec{r},E,\vec{\Omega},t)}{\partial t}+\vec{\Omega}\bullet\vec{\nabla}\psi(\vec{r},E,\vec{\Omega},t)+\Sigma_t(\vec{r},E,t)\psi(\vec{r},E,\vec{\Omega},t)=$$

$$S_{ext}(\vec{r},E,\vec{\Omega},t)+\int_0^\infty\int_{4\pi}\Sigma_s(\vec{r},E'\to E,\vec{\Omega}'\to\vec{\Omega},t)\psi(\vec{r},E',\vec{\Omega}',t)\cdot d\Omega'\cdot dE' \qquad \textbf{(1)}$$

$$+\frac{\chi(E)}{4\pi}\int_0^\infty\int_{4\pi}\nu(\vec{r},E',t)\Sigma_f(\vec{r},E',t)\psi(\vec{r},E',\vec{\Omega}',t)\cdot d\Omega'\cdot dE'$$

The terms appearing in Eq. (1) are explained below:

- $\frac{1}{v(E)}\frac{\partial\psi(\vec{r},E,\vec{\Omega},t)}{\partial t}$ is the rate of change of the neutron population with respect to time;
- $\vec{\Omega}\bullet\vec{\nabla}\psi(\vec{r},E,\vec{\Omega},t)$ is the net out-leakage rate;
- $\Sigma_t(\vec{r},E,t)\psi(\vec{r},E,\vec{\Omega},t)$ is the total collision rate;
- $S_{ext}(\vec{r},E,\vec{\Omega},t)$ is the total external neutron source per unit volume and time;
- $\int_0^\infty\int_{4\pi}\Sigma_s(\vec{r},E'\to E,\vec{\Omega}'\to\vec{\Omega},t)\psi(\vec{r},E',\vec{\Omega}',t)\cdot d\Omega'\cdot dE'$ is the total in-scattering rate;
- $\frac{\chi(E)}{4\pi}\int_0^\infty\int_{4\pi}\nu(\vec{r},E',t)\Sigma_f(\vec{r},E',t)\psi(\vec{r},E',\vec{\Omega}',t)\cdot d\Omega'\cdot dE'$ is the total neutron fission rate.

Note that terms are presented as functions of position, energy, solid angle and time in order to preserve the general nature of the equation. Terms within these groupings are further clarified:

- $v(E)$ is the neutron speed;
- $\psi(\vec{r}, E, \vec{\Omega}, t)$ is the time dependent angular flux;
- $\vec{\Omega}$ is the solid angle;
- $\Sigma_t(\vec{r}, E, t)$ is the total macroscopic collision cross-section
- $\Sigma_s(\vec{r}, E' \rightarrow E, \vec{\Omega}' \rightarrow \vec{\Omega}, t)$ is the total macroscopic scattering cross-section
- $\chi(E)$ is the fission neutron energy distribution
- $\nu(\vec{r}, E', t)$ is the average number of neutrons produced per fission reaction
- $\Sigma_f(\vec{r}, E', t)$ is the macroscopic fission cross-section

The neutron transport equation is rarely solved in its exact form. To do so requires accurate angularly-dependent cross sections for transitions to and from different neutron energies, as well as spatial scattering, absorption and fission. This requirement is excessive and very often unnecessary. Neutron cross-sections are rarely dependent upon incident angle, especially at the energies encountered in a reactor. Further, in reactors where neutrons do not thermalize significantly, a single-energy assumption may be made. While this is far from strictly accurate, it sometimes suffices to drastically reduce the number of computations required while providing sufficiently accurate results (Lamarsh 2001). If assumptions of no angular or energy dependence are made, as well as several minor substitutions, the neutron transport equation presented above can be reformulated in terms of a scalar neutron flux:

$$\frac{1}{v}\frac{\partial \phi(\vec{r},t)}{\partial t} + \vec{\nabla} \bullet \vec{J}(\vec{r},t) + \Sigma_a(\vec{r},t)\phi(\vec{r},t) = S_{tot}(\vec{r},t) \quad \textbf{(2)}$$

Several terms change slightly in the transition from Eq. (1) to Eq. (2), which is known simply as the equation of continuity. The Greek letter ψ representing angular flux is changed to ϕ, representing scalar flux. The term $\vec{J}(\vec{r},t)$ represents the neutron current, the Cartesian components of which are the directional neutron gain/loss through the x, y and z faces of a cubic control volume. This neutron current term accounts for in and out-scattering in the control volume, so the total collision rate in Eq. (1) reduces to the total neutron absorption rate, $\Sigma_a(\vec{r},t)\phi(\vec{r},t)$, where $\Sigma_a(\vec{r},t)$ is the macroscopic neutron absorption cross-section. Neutron fission and external sources are combined in the total source term, $S_{tot}(\vec{r},t)$. With these substitutions, numerical solution becomes markedly easier to implement. However, some constitutive equation relating the neutron current to the scalar flux must be formulated in order to reduce the number of unknowns in the equation.

3.1.2 Neutron Diffusion Equation

The constitutive relation most commonly used in replacing the neutron current term in Eq. (2) with some function of the scalar flux is Fick's law. Fick's law was originally used to account for chemical diffusion, where the original form of the law stated that solutes diffuse from regions of greater concentration to regions of lesser concentration (Lamarsh, 2001). Neutrons in a reactor often closely approximate this behavior. Stated mathematically, Fick's law is:

$$\vec{J}(\vec{r},t) = -D(\vec{r},t)\vec{\nabla}\phi(\vec{r},t) \quad \textbf{(3)}$$

In Eq. (3), the new term $D(\vec{r},t)$ is known as the diffusion coefficient. Its value may be approximated by:

$$D = \frac{1}{3 \cdot \Sigma_s \cdot (1-\mu)} \quad \textbf{(4)}$$

In Eq. (4), Σ_s is the macroscopic scattering cross-section and μ may be computed by a simple formula varying inversely with atomic number. As a consequence it is accurate to take μ as zero when performing diffusion calculations in a transuranic material.

Fick's law is not an exact relation, and is invalid under the following circumstances (Lamarsh, 2001) :

- In a medium that strongly absorbs neutrons;
- Within approximately three mean paths of a surface or neutron source;
- When neutron scattering is strongly anisotropic.

Substituting Fick's law into Eq. (2) gives the time-dependent neutron diffusion equation:

$$\frac{1}{v}\frac{\partial\phi(\vec{r},t)}{\partial t} - \vec{\nabla}\bullet D(\vec{r},t)\vec{\nabla}\phi(\vec{r},t) + \Sigma_a(\vec{r},t)\phi(\vec{r},t) = S_{tot}(\vec{r},t) \quad \textbf{(5)}$$

In the absence of external neutron sources in a fissioning medium, $S_{tot}(\vec{r},t)$ becomes $\upsilon\cdot\Sigma_f(\vec{r},t)\phi(\vec{r},t)$, and Eq. (5) may be rewritten:

$$\frac{1}{v}\frac{\partial\phi(\vec{r},t)}{\partial t} - \vec{\nabla}\bullet D(\vec{r},t)\vec{\nabla}\phi(\vec{r},t) + \Sigma_a(\vec{r},t)\phi(\vec{r},t) - \upsilon\cdot\Sigma_f(\vec{r},t)\phi(\vec{r},t) = 0 \quad \textbf{(6)}$$

Eq. (6) is a parabolic partial differential equation for which it is simple to obtain a variational form for use in an implementation of the finite element method.

3.1.3 Neutron Point Kinetics

A different treatment of Eq. (2) yields analytical solutions for time-dependent behavior (Hetrick, 1971); (Ott and Neuhold, 1985); (Lamarsh, 2001); (Ford, *et al.* 2003). This result is predicated upon treating the reactor as a point in space. This assumption typically introduces significant errors; however, in the case of very small reactors (i.e., SPR and Godiva reactors), it provides excellent results and has been used to describe reactor behavior accurately for decades. The governing equations for point-reactor neutron kinetics and an explanation of terms are presented here:

$$\frac{dP(t)}{dt} = \frac{\left(\rho(t) - \gamma_D \beta_D - \gamma_r \beta_r\right)}{\ell} P(t) + \sum_i \gamma_i \lambda_i C_i(t) + \sum_i \gamma_j \lambda_j C_j(t) + \sum_k \gamma_k S_k(t) \quad \textbf{(7)}$$

$$\frac{dC_i(t)}{dt} = \frac{\beta_i P(t)}{\ell} - \lambda_i C_i \quad \textbf{(8)}$$

$$\frac{dC_j(t)}{dt} = \frac{\beta_j P(t)}{\ell} - \lambda_j C_j \quad \textbf{(9)}$$

The reactivity relationship may be described by the following equations:

$$\rho(t) = \rho_{input}(t) - \alpha_T \Delta T_f(t) \quad \textbf{(10)}$$

$$\Delta T_f = \frac{E(t)}{CpM} \quad \textbf{(11)}$$

$$E(t) = \int (P(t) - Heatloss(t))dt \quad \textbf{(12)}$$

The terms used within these equations are summarized below.

- P(t) is the fission power at time t,
- ρ is the system reactivity, defined as $\frac{k-1}{k}$;
- k is the effective multiplication factor,
- $\rho_{input}(t)$ is the reactivity input,
- α_T is the negative temperature reactivity feedback coefficient,
- ΔT_f is the change in fuel temperature,

- ℓ is the effective prompt neutron generation time (i.e., neutron lifetime/k),
- $C_i(t)$ is the population at time t of the i^{th} delayed neutron precursor group,
- β_i is the fraction of all non-source neutrons produced that result from the decay of the i^{th} delayed neutron precursor group,
- λ_i is the decay constant of the i^{th} delayed neutron precursor group,
- γ_i is the relative effectiveness of the i^{th} group delayed neutrons producing fission,
- $C_j(t)$ is the population at time t of the j^{th} γ reflector group,
- β_j is the effective fraction of all leakage neutrons returned in the j^{th} group due to refection from the reactor room walls, experiments, and/or other hardware in the vicinity of the reactor,
- λ_j is the decay constant of the j^{th} reflector group,
- γ_j is the relative effectiveness of the j^{th} reflector group,
- $S_k(t)$ is the rate at which neutrons are added to the system by the k^{th} source, which is independent or can be treated as independent of the fuel system (e.g., isotopic sources, accelerator-induced sources), and
- γ_k is the relative effectiveness of neutrons produced by the k^{th} source.

Next, we make several assumptions typical to modeling fast burst reactors: reflected neutrons may be ignored, changes in delayed neutron precursors may be ignored, and that only a fixed neutron source is present. Under these assumptions, the above equations reduce to:

$$\frac{dP(t)}{dt} = \frac{(\rho(t) - \beta)}{\ell} P(t) + S \qquad \textbf{(13)}$$

For excursions in which the FWHM duration of the power burst is significantly less than the half-lives of the delayed neutron precursors, the equations simplify to the Nordheim-Fuchs model (Hetrick, 1971). Since the half-life of the fastest decaying delayed neutron group is ~0.230 s (Lamarsh, 2001), and the FWHM of a typical SPR III pulse is ~100 μs (see Table 2.2), this model is highly useful in predicting the behavior of fast burst reactors such as SPR II and SPR III.

$$\frac{dP(t)}{dt} = \frac{(\rho(t) - \beta)}{\ell} P(t) \tag{14}$$

$$\rho(t) = \rho_i(t) - \alpha_T T(t) \tag{15}$$

where:

- ρ_i is the reactivity added at time *t*, and
- α_T = negative temperature reactivity feedback coefficient (i.e., reactivity change per unit of fuel temperature change).

It is useful to summarize some nomenclature unique to reactor kinetics at this point. In order to quantify reactivity more intuitively, it is defined in "dollars and cents." These units correspond directly to the term β defined above: one dollar corresponds to a reactivity value of exactly β. One cent is of course one-hundredth of β.

The physical significance of these terms is as follows: if one adds exactly one dollar of reactivity to a precisely critical (k = 1) nuclear system, it is said to become

prompt supercritical. This means the reactor is now critical on prompt neutrons (i.e., neutrons produced during a fission reaction) alone. An essential characteristic of a prompt supercritical reactor is that its neutron population is multiplying at a rate uncontrollable by active systems. More specifically, the *period* of the reactor (one period being the time required for the system neutron population to increase by a factor of the natural logarithm base) becomes extremely small. Any reactivity addition past one dollar reduces the period further.

An assumption of adiabatic heating is valid on time scales hundreds of times that of a SPR III pulse duration. This follows from observation of the time-dependent behavior of large temperature gradients in metallic solids, *i.e.* it requires approximately ten seconds for gradients spanning the range of uranium's melting point to "flatten." In a sea-level air atmosphere with no active cooling, other modes of heat transfer may be assumed negligible compared to internal heat conduction. This adiabatic assumption can be used to define the feedback in terms of $\mu_f E(t)$, where:

$$E(t) = \int_0^t P(t')dt' \tag{16}$$

The fission energy yield E(t) is related to the temperature rise by:

$$E(t) = MCp\Delta T(t) \tag{17}$$

where *M* is the fuel mass and *Cp* is the fuel specific heat.

The energy feedback coefficient is related to the temperature feedback coefficient by:

$$\mu_f = \frac{\alpha_t}{CpM} \tag{18}$$

For fixed reactivity additions ρ_0 above prompt supercritical (> \$1.00), and adiabatic heating, the resulting fission energy yield and maximum temperature rise may be expressed as:

$$E(\infty) = \frac{2(\rho_0 - \beta)}{\mu_f} \tag{19}$$

$$\Delta T(\infty) = \frac{2(\rho_0 - \beta)}{\alpha_t} \tag{20}$$

These kinetics equations have been used successfully to describe the behavior of fast burst reactors and assemblies (Ford, *et al.* 2003).

3.2 LINEAR ELASTICITY

Here we present a brief review of the basic equations for an elastic continuum, derived in large part from Graff's text on wave propagation in elastic solids (Graff, 1975). The equations shall be presented in Cartesian coordinates, with a summary of the spherical and cylindrical coordinate equations of interest to the problems at hand.

3.2.1 Basic Equations

We now present the equations governing linear isotropic elastic solids. First, we define the infinitesimal strain and rotation tensors, respectively:

$$e_{ij} = \frac{1}{2} \cdot (u_{i,j} + u_{j,i})$$
$$\omega_{ij} = \frac{1}{2} \cdot (u_{i,j} - u_{j,i}) \quad \textbf{(21)}$$

While the mathematical derivations are not reproduced here, qualitatively these definitions derive from the displacement and deformation of a continuum of volume V to volume V'. For a point P_0 in the undeformed, undisplaced continuum, its resulting kinematics are governed by the local strain-gradient field, and its motion is a combination of the strain and rotation tensors.

Given a continuum subject to various forces, resulting tractive contact forces will act on arbitrary surface elements within the body. The traction vector is given by:

$$\vec{t} = t_j i_j \quad \textbf{(22)}$$

where i are normal forces acting upon surfaces of an arbitrary volume element. The traction components t_j serve to define the Cartesian stress tensor:

$$t_i = \tau_{ij} n_j \quad \textbf{(23)}$$

where n_j are basis vectors normal to surface j.

Next we present the equations for conservation of mass, momentum, moment of momentum and energy.

$$
\begin{aligned}
&\textit{mass}: && \frac{\partial \rho}{\partial t} + (\rho \dot{u}_{i,i}) = 0 \\
&\textit{momentum}: && \tau_{ji,i} + \rho f_i = \rho \ddot{u}_i \\
&\textit{moment of momentum}: && \tau_{ij} = \tau_{ji} \\
&\textit{energy}: && \rho \dot{\varepsilon} = \tau_{ji} \dot{\varepsilon}_{ij}
\end{aligned}
\qquad \textbf{(24)}
$$

where:

- is mass density;
- f_i is body force;
- is internal energy per unit mass.

Note also that:

$$\dot{\varepsilon}_{ji} = \dot{\omega}_{ij} - \dot{u}_{i,j} \qquad \textbf{(25)}$$

These conservation equations constitute a system of eight equations with thirteen unknowns. Information on the material body is required to make the equations determinate.

Constitutive equations relate strain to stress. Both Green and Cauchy developed formulations for ideal elastic bodies. Green's method will be presented here.

The formulation depends on the perfect elasticity of the body. This means there are no dissipative mechanisms, and the constitutive equations must be derivable from an internal energy function of the strain. Defining this function, substituting the conservation of energy relation, and expanding the internal energy function in a power series of the strain yields:

$$\tau_{ij} = a_{ijkl}\varepsilon_{ij}\varepsilon_{kl} \tag{26}$$

Observing symmetry with respect to *ij* and *kl* in Green's formulation reduces the 81 constants a_{ijkl} to 21 for general anisotropy.

Finally, by assuming a homogeneous, isotropic material (such as U-10 Mo alloy), the number of elastic constants reduces from 21 to 2. Note these two numbers correspond to the general anisotropic case and the isotropic case, respectively. The equations for linear isotropic elasticity are:

$$\begin{aligned}
&\tau_{ij,j} + \rho f_i = \rho \ddot{u}_i \\
&\tau_{ij} = \lambda \varepsilon_{kk}\delta_{ij} + 2\mu\varepsilon_{ij} \\
&\varepsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}) \\
&\omega_{ij} = \frac{1}{2}(u_{i,j} - u_{j,i})
\end{aligned} \tag{27}$$

The coefficients λ and μ are known as Lame's constants. The modulus of elasticity, Poisson's ration, and the bulk material modulus may all be expressed in terms of these two coefficients.

Utilizing the stress-strain and strain-displacement relations in Eq. (21) and Eq. (25), we can express the momentum equation in vector form:

$$(\lambda + \mu)\nabla\nabla\cdot\vec{u} + \mu\nabla^2\vec{u} + \rho f = \rho\ddot{u} \tag{28}$$

3.2.2 Cylindrical Coordinates

We present the elastic equations in the cylindrical coordinate system applicable to modeling SPR reactors in the more familiar differential form:

$$
\begin{aligned}
e_{rr} &= \frac{\partial u_r}{\partial r} \\
e_{zz} &= \frac{\partial u_z}{\partial z} \\
e_{\theta\theta} &= \frac{1}{r}\frac{\partial u_\theta}{\partial \theta} + \frac{u_r}{r} \\
e_{r\theta} &= \frac{1}{2}\left(\frac{1}{r}\frac{\partial u_r}{\partial \theta} + \frac{\partial u_\theta}{\partial r} - \frac{u_\theta}{r}\right) \\
e_{rz} &= \frac{1}{2}\left(\frac{\partial u_z}{\partial r} + \frac{\partial u_r}{\partial z}\right) \\
e_{\theta z} &= \frac{1}{2}\left(\frac{\partial u_\theta}{\partial z} + \frac{1}{r}\frac{\partial u_z}{\partial \theta}\right) \\
\frac{\partial \tau_{rr}}{\partial r} + \frac{1}{r}\frac{\partial \tau_{r\theta}}{\partial \theta} + \frac{\partial \tau_{rz}}{\partial z} + \frac{\tau_{rr} - \tau_{\theta\theta}}{r} + \rho f_r &= \rho \ddot{u}_r \\
\frac{\partial \tau_{r\theta}}{\partial r} + \frac{1}{r}\frac{\partial \tau_{\theta\theta}}{\partial \theta} + \frac{\partial \tau_{\theta z}}{\partial z} + \frac{2}{r}\tau_{r\theta} + \rho f_\theta &= \rho \ddot{u}_\theta \\
\frac{\partial \tau_{rz}}{\partial r} + \frac{1}{r}\frac{\partial \tau_{\theta z}}{\partial \theta} + \frac{\partial \tau_{zz}}{\partial z} + \frac{1}{r}\tau_{rz} + \rho f_z &= \rho \ddot{u}_z
\end{aligned}
\qquad \textbf{(29)}
$$

Note that heat effects have been neglected from the basic elasticity equations in cylindrical coordinates (*i.e.,* thermal expansion and material heating due to displacement are ignored). Adding a conservation of entropy equation to the conservation equations leading to Eq. (29) allows us to account for the effects of temperature gradients severe enough to cause interaction between heat and mechanical vibration. These interactions are known as thermoelastic waves. It is worthwhile to present the differential form of these equations, even though the simplifying Duhamel-Neumann analogy (Fung and Tong, 2001) is applied in the practical numerical solution. Initial numerical work was

based exclusively on the differential form of the dynamic thermoelastic equation (Wimmet, 1992); (Wilson, 2004).

We present the appropriate dynamic equation in cylindrical geometry with angular symmetry applicable to our system (Wimmet, 1992):

$$\frac{(1-\nu)E}{(1-2\nu)(1+\nu)}\cdot D^2(u)+\frac{E}{2(1+\nu)}\cdot\frac{\partial^2 u}{\partial z^2}+\frac{E}{2(1-2\nu)(1+\nu)}\cdot\frac{\partial^2 v}{\partial r\partial z}$$
$$-\frac{E}{(1-2\nu)}\cdot\beta\cdot\frac{\partial T}{\partial r}=\rho\cdot\frac{\partial^2 u}{\partial t^2} \qquad \textbf{(30)}$$

The terms used in this equation are as follows:

- u is outward radial displacement;
- v is axial displacement;
- is the coefficient of thermal expansion;
- T is T(r,t), the fuel temperature rise;
- t is time;
- $D^2(u)$ is: $\frac{\partial^2 u}{\partial r^2}+\frac{\partial u}{r\partial r}+\frac{u}{r^2}$;
- E is Young's modulus ($\frac{\mu(3\lambda+2\mu)}{(\lambda+\mu)}$);
- is Poisson's ratio ($\frac{\frac{1}{2}\lambda}{(\lambda+\mu)}$);
- is fuel mass density.

The radial, tangential, axial, and shear stress components, respectively, under angular symmetry, in terms of strain, are presented:

$$\begin{aligned}
\frac{\tau_r}{E} &= \frac{(1-\nu)}{(1-2\nu)(1+\nu)}\cdot\frac{\partial u}{\partial r} + \frac{\nu}{(1-2\nu)(1+\nu)}\cdot\left(\frac{u}{r}+\frac{\partial v}{\partial z}\right) - \frac{\beta T}{(1-2\nu)} \\
\frac{\tau_\theta}{E} &= \frac{(1-\nu)}{(1-2\nu)(1+\nu)}\cdot\frac{u}{r} + \frac{\nu}{(1-2\nu)(1+\nu)}\cdot\left(\frac{\partial u}{\partial r}+\frac{\partial v}{\partial z}\right) - \frac{\beta T}{(1-2\nu)} \\
\frac{\tau_z}{E} &= \frac{(1-\nu)}{(1-2\nu)(1+\nu)}\cdot\frac{\partial v}{\partial z} + \frac{\nu}{(1-2\nu)(1+\nu)}\cdot\left(\frac{u}{r}+\frac{\partial u}{\partial r}\right) - \frac{\beta T}{(1-2\nu)} \\
\frac{\tau_{rz}}{E} &= \frac{1}{2(1+\nu)}\cdot\left(\frac{\partial u}{\partial z}+\frac{\partial v}{\partial r}\right)
\end{aligned} \qquad \textbf{(31)}$$

Note the dynamic equation is in two separate displacement variables: one for axial (z) displacement, and the other for radial (r) displacement. While this is solvable in two dimensions using numerical techniques and the appropriate boundary and initial conditions, a one-dimensional solution can approximate fairly closely the behavior of SPR reactors under pulse conditions (Wimmet, 1992); (Wilson, 2004).

Recall that SPR III was specifically designed not only with segmented fuel rings, but in a way that reduces contact between successive fuel rings to only ~25% of their surface area. Ring-to-ring axial wave propagation is minimized to the point of negligibility. This permits certain assumptions based on the geometry of *individual* rings: i.e., where the axial dimension is actually much smaller than the radial.

In his analytical treatment of SPR II pulses, Wimmet presents three approximations utilizing the assumption of zero axial stress wave propagation. Based on his results (Wimmet 1992), the coupled method-of-lines/Monte Carlo method implemented in this work utilized a disk approximation in which axial and shear stress are both assumed to be negligible:

$$\tau_z = 0$$
$$\tau_{rz} = 0 \tag{32}$$

This permits manipulation of the stress components to eliminate both axial displacement in our final dynamic equation, and to obtain an average axial displacement in terms of radial displacement.

Given zero axial stress, we obtain from Eq. (27):

$$\frac{\partial v}{\partial z} = -\frac{\nu}{(1-\nu)} \cdot (\frac{\partial u}{\partial r} + \frac{u}{r}) + \frac{(1+\nu)}{(1-\nu)} \cdot \beta T$$
$$\frac{\partial^2 v}{\partial r \partial z} = -\frac{\nu}{(1-\nu)} \cdot D^2(u) + \frac{(1+\nu)}{(1-\nu)} \cdot \beta \cdot \frac{\partial T}{\partial r} \tag{33}$$

Given zero shear stress, we obtain from Eq. (31):

$$\frac{\partial u}{\partial z} = -\frac{\partial v}{\partial r}$$
$$\frac{\partial^2 u}{\partial z^2} = -\frac{\partial^2 v}{\partial r \partial v} \tag{34}$$

Combining these results gives a dynamic equation in only the radial displacement variable, reducing Eq. (30) to:

$$D^2(u) = (1+\nu) \cdot \beta \cdot \frac{\partial T}{\partial r} + \frac{(1-\nu^2)\rho}{E} \cdot \frac{\partial^2 u}{\partial t^2} \tag{35}$$

This is the displacement equation used in modeling SPR behavior in the method-of-lines/Monte Carlo implementation presented later in this work. Its solution gives radial displacement directly; the zero shear stress condition is manipulated to obtain an approximation of axial displacement. The coefficient preceding the second order time derivative is the ($1/c^2$) wave velocity term. Further treatment by Wimmet results in a slightly superior wave velocity, which is presented here:

$$c^2 = \left[\frac{(2-\nu)E}{2(1-\nu^2)\rho} \right]^{\frac{1}{2}} \tag{36}$$

Finally, we observe that Eq. (31), the dynamic equation, is a second order hyperbolic partial differential equation. The initial values of displacement and its time derivative are zero, and require two boundary conditions for a unique solution. They are, very simply, that radial stress goes to zero at the interior and exterior radial boundaries *a* and *b*:

$$\tau_r(a) = \tau_r(b) = 0 \tag{37}$$

It is important to note that these boundary conditions are the driving force of most radial oscillation occurring in the fuel. By observing Eq. (35), the dynamic displacement equation, one might deduce that there would be *no* movement in the fuel in the absence of a temperature gradient. This is clearly and intuitively not the case. While large temperature gradients affect the resulting oscillation, a steep, flat temperature transient will cause a similar oscillation. By observing in Eq. (31) that the radial stress equations

have a temperature term, instead of a temperature derivative term, we see that fuel movement is possible in the absence of a temperature gradient.

Finally, as stated previously, the differential form of the dynamic thermoelastic equation in cylindrical coordinates is not explicitly used in the finite element implementation. Observing the dynamic differential equation provides some insight into the physical behavior of the system, as well as the differences between the SPR reactors and the solid hollow right cylinder actually modeled. Further, significant initial computational work utilizes the one-dimensional approximation given in Eq. (35).

In this work, a simple, general axisymmetric elasticity code is developed and the appropriate surface tractions and internal body forces are applied, providing a full two-dimensional solution. This development is described later in this dissertation.

3.2.3 Spherical Coordinates

The governing thermoelastic equation in differential form in spherical coordinates with angular symmetry is presented here without derivation:

$$\frac{\partial^2 u}{\partial r^2}+\frac{2}{r}\frac{\partial u}{\partial r}-\frac{2u}{r^2}=\frac{1+\nu}{1-\nu}\beta\frac{\partial T}{\partial r}+\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} \tag{38}$$

$$c=\left[\frac{(1-\nu)E}{(1+\nu)(1-2\nu)\rho}\right]^{1/2}$$

It is worth noting that this equation is a hyperbolic partial differential equation that can be put into a useful variational form by utilizing a transformation of the dependent variable common to spherical coordinates. Namely,

$$w(r,t) = u(r,t) \cdot r \tag{39}$$

This substitution into Eq. (38) eliminates the first order spatial derivative and its attendant singularity at $r = 0$. Boundary condition changes under this transformation further eliminate the singularity in the zero order term. It is similarly used in resolving the Laplacian term in the neutron diffusion equation in spherical coordinates with angular symmetry, as will be explained shortly.

3.2.4 Duhamel-Neumann Analogy

As stated above, a well-established solution method for plane and axisymmetric elasticity exists. The principal of virtual work provides the theoretical basis for the finite element model, *e.g.* (Becker, *et al.*, 1981); (Zienkiewicz, *et al.*, 2000); (Fung and Tong, 2001); (Reddy 2006). In the case at hand, the only surface tractions and body forces considered result from fission heating of the solid body. The Duhamel-Neumann analogy makes calculation of these tractions and body forces simple. Fung and Tong's text summarizes it succinctly:

> "[T]he effect of a temperature change θ is equivalent to replacing the body force X_i in Navier's equations by X_i-$\beta\theta$, and to substituting for the surface tractions g_i by $g_i + \beta\nu_i\theta$. Thus the displacements *u1*, *u2* and *u3* produced by a temperature

change θ are the same as those produced by the body forces $-\beta\theta_{,i}$ and the normal tractions $\beta\theta$ acting on the surface of a body of the same shape but throughout which the temperature is normal." (Fung and Tong, 2001)

In the statement above, β is defined as $\frac{E \cdot \alpha}{1 - 2 \cdot \nu}$, where E is Young's modulus, α is the linear coefficient of expansion, and ν is Poisson's ratio. This makes each surface traction and body force in a finite element mesh a function of either the local temperature rise or the spatial derivative thereof.

3.3 Overview of the Finite Element Method

While a thorough explanation of the finite element method is beyond the scope of this work, it is anticipated that some readers of this report may be unfamiliar with the method. A basic introduction to the steps involved in implementing a finite element solution is helpful in understanding the codes developed herein. A number of excellent introductions to this topic exist. In particular, we draw heavily on Becker, Carey and Oden's introductory text (Becker, *et al.*, 1981) for a succinct explanation of the Galerkin approximation, variational formulation, and other finite element basics, and the section on plane and axisymmetric elasticity in Fung and Tong's computational solid mechanics text (Fung and Tong, 2001).

3.3.1 Variational Formulation

The variational, or weak form of a differential equation is obtained by multiplying the classical equation by an admissible test function and using integration by parts to

reduce the order of differentiation of second (or higher) order terms. For example, if one were given the classical boundary value problem:

$$\begin{aligned} &-u'' + u = x, \quad 0 < x < 1 \\ &u(0) = u(1) = 0 \end{aligned} \tag{40}$$

One obtains the weak form of this equation by integrating both sides over the domain, multiplying by a well-behaved test function, and integrating by parts:

$$\begin{aligned} &\int_0^1 (-u'' + u)dx = \int_0^1 x dx \\ &\int_0^1 (-u'' + u)\cdot v dx = \int_0^1 x \cdot v dx \\ &\int_0^1 (-u' \cdot v' + u \cdot v - x \cdot v)\cdot dx = 0 \end{aligned} \tag{41}$$

Eq. (41) gives a simple example in one dimension of obtaining the variational form of a differential equation. There are several reasons to do so. By moving an order of differentiation of the solution function onto the test function, we weaken the conditions for finding a solution and its derivatives by increasing the class of admissible test functions. This permits finding solutions to differential equations where the solution is very irregular. Further, imposing certain restrictions on the test function *v*(x) permits use of a Galerkin approximation to the solution, which is the foundation of the Galerkin finite element method (Becker, *et al.*, 1981).

3.3.2 Galerkin Approximation

The admissible test functions belong to the H_0^1 linear infinite-dimensional function space. These functions vanish on the problem endpoints and are square-integrable. They may be represented as linear combinations of an infinite set of functions ϕ_i:

$$v(x) = \sum_{i=1}^{\infty} \beta_i \phi_i(x) \tag{42}$$

In Eq. (42), the β_i are constants and the series is assumed convergent. The set of functions ϕ is said to provide a basis for H_0^1 and the individual functions ϕ_i are called basis functions (Becker, *et al.*, 1981).

If only a finite number of terms are taken in Eq. (42), we obtain an approximation of v(x):

$$v_N(x) = \sum_{i=1}^{N} \beta_i \phi_i(x) \tag{43}$$

The N basis functions define an N-dimensional subspace $H_0^{\langle N \rangle}$ of H_0^1. Galerkin's method constructs an approximate solution to Eq. (41) in this finite dimensional subspace instead of on the infinite dimensional space. The approximate solution appears as:

$$u_N(x) = \sum_{i=1}^{N} \alpha_i \phi_i(x) \tag{44}$$

The variational statement of this approximate solution becomes finding u_N such that

$$\int_0^1 (u_N{}' v_N{}' + u_N v_N) \cdot dx = \int_0^1 x v_N dx \tag{45}$$

for all v_N on $H_0^{\langle N \rangle}$. This can be rewritten by defining the following:

$$K_{ij} = \int_0^1 \phi_i'(x)\phi_j'(x) + \phi_i(x)\phi_j(x)\, dx \tag{46}$$

$$F_i = \int_0^1 x\phi_i(x)\, dx \tag{47}$$

Inserting Eq. (46) and (47) into our finite variational problem gives:

$$\sum_{j=1}^{N} K_{ij}\alpha_j = F_i \tag{48}$$

$$i = 1, 2, \ldots N$$

This derivation provides all the tools we need to construct an approximation of the continuous solution *u*(x) except for an appropriate finite subset of basis functions. The finite element method provides a systematic framework for constructing these basis functions (Becker, *et al.*, 1981). Eq. (48) gives the general form of the matrix problem resulting from a finite element method implementation.

3.3.3 Element Discretization and Local Shape Functions

In order to generate these global basis functions, we discretize the problem domain into finite elements and locally define simple shape functions. Each element has a

certain number of nodes based on the order of shape functions employed. The solution will be determined directly at these nodes, and interpolated elsewhere. These shape functions act to generate the element contribution to the global basis. Summing together the element contributions in a systematic way generates the basis functions. Referring to Eqs. (46-48), the element contributions are defined for the model variational problem stated in Eq. (45) as:

$$k_{ij}^{e} = \int_{\Omega_e} \psi_i^{'}(x)\psi_j^{'}(x) + \psi_i\ (x)\psi_j\,(x)\,dx$$
$$f_i^{e} = \int_{\Omega_e} x\,\psi_i(x)\,dx \qquad \textbf{(49)}$$

The square matrix *K* defined in Eq. (48) is called the stiffness matrix. It is created by assembling the element contributions defined in Eq. (49). The vector *F* is the load vector, and it is similarly assembled. The functions $\psi_i(x)$ in the integrands of Eq. (49) are the local shape functions. It is worth noting that in second order partial differential equations lacking a first order derivative, the stiffness matrix *K* is symmetric. Due to the assembly process, it also tends to be sparse. These two facts play prominently in reducing the number of integrals evaluated to populate the matrix.

The assembly process for the stiffness matrix and load vector will not be described in detail here, as it is somewhat beyond the scope of a fast introduction to the basic concepts of finite elements. The material in Chapter 4, and the finite element codes themselves and their attendant documentation in Appendices A through D should make the assembly process for linear elements in one and two dimensions fairly clear.

For a steady-state elliptic PDE such as the one described in Eq. (40), implementing the finite element method reduces finding a solution function to solving a simultaneous linear system of equations for a vector of unknowns:

$$Ku = F \tag{50}$$

In Eq. (50), K and F are the stiffness matrix and load vector described above. The vector u is the vector of unknown values at the element nodes for whichever field variable the differential equation describes; in the case of neutron diffusion it is the scalar neutron flux, and in elasticity it is the displacement from a reference state.

Solving the system of equations for the vector of unknowns described in Eq. (50) is a significant problem in and of itself. While one-dimensional problems are fairly trivial, two-dimensional problems can involve many thousands of degrees of freedom, and three-dimensional problems can involve millions. Gauss-Jordan elimination provides a sure way of solving linear systems of equations, but fails to take advantage of the helpful symmetry and sparseness of the stiffness matrix. The codes developed in this work utilize Gauss-Jordan elimination for the sake of simplicity, but other options, particularly banded solvers, do the job more efficiently and offer significant decreases in storage requirements (Becker, *et al.*, 1981); (Zienkiewicz, *et al.*, 2000); (Fung and Tong, 2001).

3.3.4 Coordinate Transform

A complete explanation of coordinate transforms between meshes in finite element implementations is more complicated than necessary for the discussion at hand. A brief qualitative explanation is given here because of the importance of coordinate transforms in the two-dimensional code developed in this dissertation. In-depth analysis

of coordinate transforms may be found in any good finite element text (Becker, *et al.*, 1981); (Zienkiewicz, *et al.*, 2000).

An important strength of the finite element method is its ability to discretize complicated or irregular domain shapes. Finite element mesh generation usually does not generate many elements of exactly the same shape and size; instead meshes are generated using one basic element type of different dimensions and orientations. In one dimension it is trivial to implement this, as it involves simply lengthening and shortening a line. However, in two and three dimensions it is inconvenient and inefficient to evaluate integrals on many different domains. Instead, a master element is defined where numerical integration is easy to implement. Every element in the mesh is then mapped onto this master element for integral evaluation. The steps of this coordinate transform are roughly as follows:

- Select and define a master element;
- Define shape functions on the master element;
- Use the element shape functions to map the element nodal coordinates on the mesh coordinate system to the corresponding nodal coordinates on the master element;
- Use this mapping to obtain similar transformations for infinitesimals on the mesh coordinates to the master element coordinates;
- Define and evaluate a Jacobian matrix populated by derivatives of the mesh coordinates with respect to the master element coordinates;
- Obtain the determinant of this matrix;

- Use this determinant, simply called the Jacobian, to obtain expressions for the derivatives of the master element coordinates with respect to the mesh coordinates.

These steps give expressions for the coordinates of each system in terms of the other, as well as their derivatives with respect to one another. These expressions may be used in a chain rule under the integrand of each stiffness and load integral to transform from one coordinate system to the other.

3.3.5 Numerical Integration

Numerical integration of the element contributions to the global stiffness matrix and load vector is as important to an implementation of the finite element method as a good linear solver. The coordinate transformation described previously greatly simplifies the numerical integration process: master element dimensions can be chosen specifically to eliminate extra coefficients, and shape functions may be selected so that an appropriate quadrature rule may integrate them exactly.

The quadrature rule of choice in any finite element code utilizing coordinate transforms (*i.e.*, any problem in two or more spatial dimensions) is Gaussian quadrature. A Gauss rule of order N exactly integrates polynomials of degree $2N - 1$.

If coordinate transformation requires more computational effort than using a slightly higher order Newton-Cotes quadrature rule, doing so may be excused. Generally this is only the case in very regular meshes (*i.e.*, in one spatial dimension).

3.3.6 Time Dependence

Marching a finite element solution in time typically involves discretizing in the time dimension and using some type of differencing method. Since both codes developed in this report are time-dependent, we present algorithms for both parabolic and hyperbolic partial differential equations.

The matrix equation for parabolic partial differential equations appears as:

$$M\dot{d} + Kd = F \tag{51}$$

Compared to Eq. (50) a new matrix *M* appears as the coefficient of the first order time derivative. This is known as the mass or capacitance matrix. For parabolic equations the most commonly used algorithms are members of the generalized trapezoidal family. It consists of the following set of equations:

$$\begin{aligned} &M\, v_{n+1} + K\, d_{n+1} = F_{n+1} \\ &d_{n+1} = d_n + \Delta t \cdot v_{n+\alpha} \\ &v_{n+\alpha} = (1-\alpha) v_n + \alpha\, v_{n+1} \end{aligned} \tag{52}$$

The terms in Eq. (52) are straightforward. Vectors *d* and *v* are the field variable and its first order time derivative, respectively. The term Δt is the size of the time step. The scalar coefficient α defines which differencing method is used: if α is 0, it is forward differences, if α is 0.5 it is the trapezoidal or midpoint rule, and if α is 1 it is backward differences. In order to solve the set of equations in Eq. (52) an implementation must be chosen. The v-form implementation (Hughes, 2000) may be summarized:

- Define an initial value for vector *d*
- Solve the following set of equations for the initial value of vector *v*:

$$Mv_0 = F_0 - Kd_0 \tag{53}$$

- Define a predictor value of d_{n+1} :

$$\tilde{d}_{n+1} = d_n + (1-\alpha)\Delta t \cdot v_n \tag{54}$$

- Use the following expression to solve for the next value of *v*:

$$(M + \alpha\Delta tK)v_{n+1} = F_{n+1} - K\,\tilde{d}_{n+1} \tag{55}$$

- Finally, use the velocity vector in Eq. (55) to solve for the next *d*:

$$d_{n+1} = \tilde{d}_{n+1} + (1-\alpha)\Delta t \cdot v_{n+1} \tag{56}$$

Repeating this loop of Eq. (54), Eq. (55) and Eq. (56) advances the vector of nodal point solutions in time.

Hyperbolic partial differential equations have a similar form:

$$M\ddot{d} + C\dot{d} + Kd = F \tag{57}$$

Here the mass matrix *M* precedes a second order time derivative term and a new matrix *C*, called the viscous damping (or simply damping) matrix precedes the first order time derivative term. The most common family of methods for solving Eq. (57) is the Newmark family (Hughes, 2000). It consists of the following equations:

$$\begin{aligned} &Ma_{n+1} + Cv_{n+1} + Kd_{n+1} = F_{n+1} \\ &d_{n+1} = d_n + \Delta t \cdot v_n + \frac{1}{2}\Delta t^2[(1-2\beta)a_n + 2\beta a_{n+1}] \\ &v_{n+1} = v_n + \Delta t[(1-\gamma)a_n + \gamma a_{n+1}] \end{aligned} \tag{58}$$

A new vector *a* appears in Eq. (58). It represents the second order time derivative of the solution vector *d*. The a-form implementation of Eq. (58) is similar to the previous implementation of the parabolic case, and is given below:

- Define initial values for vectors *d* and *v*;
- Calculate an initial value for *a* using the relation:

$$Ma_0 = F - Cv_0 - Kd_0 \tag{59}$$

- Define predictors for *d* and *v*:

$$\begin{aligned} &\tilde{d}_{n+1} = d_n + \Delta t \cdot v_n + \frac{1}{2}\Delta t^2(1-2\beta)a_n \\ &\tilde{v}_{n+1} = v_n + (1-\gamma)\Delta t \cdot a_n \end{aligned} \tag{60}$$

- Solve a recursion relation for the value of *a* at the next time step:

$$(M + \gamma \Delta t C + \beta \Delta t^2 K) a_{n+1} = F_{n+1} - C \tilde{v}_{n+1} - K \tilde{d}_{n+1} \quad \textbf{(61)}$$

- Finally, use the value of a_{n+1} acquired in Eq. (61) to solve for d_{n+1} and v_{n+1}:

$$\begin{aligned} d_{n+1} &= \tilde{d}_{n+1} + \beta \Delta t^2 a_{n+1} \\ v_{n+1} &= \tilde{v}_{n+1} + \gamma \Delta t a_{n+1} \end{aligned} \quad \textbf{(62)}$$

Again, looping through the process and solving Eq. (60), Eq. (61) and Eq. (62) moves the vector of nodal point solutions *d*, as well as its time derivatives, forward in time. Different values for β and γ specify different time marching methods. The method implemented in the codes developed for this report set β to 0.25 and γ to 0.5: this is the trapezoidal rule. It is implicit and unconditionally stable (Hughes, 2000).

It is also worth noting that the damping matrix *C* is usually taken as a linear combination of the mass and stiffness matrices. It is actually taken as zero in the finite element codes in Appendices A and C: damping in the essential first oscillation period is small enough to be ignored, and experimental information on the damping of U-10 Mo is somewhat limited.

3.3.7 Boundary Conditions

It is worth noting that no part of the finite element calculation to this point requires any information about boundary conditions. It is the same for every problem solution. Boundary conditions are not implemented until after the global stiffness and mass matrices and load vector have been fully calculated.

When some linear combination of the field variable and its spatial derivatives has been prescribed, we have a general natural boundary condition (Becker, *et al.* 1981). It is of the form:

$$u'(a) = \frac{\gamma - \beta u(a)}{\alpha} \tag{63}$$

where a is the spatial coordinate at which the boundary condition is applied, and γ, β and α are coefficients which may or may not be functions of position (or even time).

Dirichlet boundary conditions are a special case of Eq. (63) where α is zero. They are of the form:

$$u(a) = \frac{\gamma}{\beta} \tag{64}$$

where once again γ and β are coefficients usually exhibiting spatial and temporal dependence. Finally a special case of Eq. (63) arises when β is zero, and only the derivative is specified on the boundary. These are called Neumann conditions:

$$u'(a) = \frac{\gamma}{\alpha} \tag{65}$$

The expressions in Eqs. (63), (64) and (65) apply directly to one-dimensional problems (where a is the spatial coordinate), but the concepts apply in two and three dimensions as well. Specifics on implementation of the boundary conditions applicable to the system being modeled herein, as well as other choices made in the finite element implementation, may be found in Chapter 4.

Chapter 4: Finite Element Development

4.1 BACKGROUND

The theoretical considerations developed in Chapter 3 give insight into the equations governing the systems whose behavior we wish to describe numerically and provide a viable method of doing so. In this chapter we explain the specific choices and background calculations made in implementing a finite element solution to these systems, including:

- Variational forms of the governing equations;
- Explicit expressions for stiffness, load and mass contributions;
- Master element type and order;
- Master element shape functions;
- Finite element mesh structure and refinement;
- Boundary conditions;
- Integration type and order;
- Time-marching scheme;
- Program logic flow.

Separate discussions are provided for the one-dimensional and two-dimensional codes. From this point forward, the one-dimensional code is referred to as *ODMain* and the two-dimensional code as *TDMain*.

4.2 ONE-DIMENSIONAL CODE

Information about the one-dimensional code is presented here without significant derivation. However, decisions made in the implementation of the finite element method will be justified where appropriate.

4.2.1 Variational Forms

The two equations being solved in *ODMain* are Eq. (6), where the Laplacian is in spherical coordinates, and Eq. (38). Both equations utilize the very important substitution in Eq. (39). This substitution eliminates the first order derivative term in both equations, and also converts the Neumann condition at the sphere center for Eq. (38) into a Dirichlet condition. As stated in Chapter 3, the presence of a first order term in the variational equation leads to asymmetric stiffness matrices. This in turn effectively doubles the number of integrals evaluated, and is to be avoided whenever possible. Here we present the variational forms of Eqs. (6) and (38), where the substitution of Eq. (39) has already occurred:

$$\int_{\Omega} \frac{1}{V D} \frac{\partial w}{\partial t} - w'v' + \left(\frac{\Sigma_a - \nu \Sigma_f}{D}\right) wv = \int_{\Omega} S_{ext} v \qquad \textbf{(66)}$$

$$\int_{\Omega} \frac{1}{c^2} \frac{\partial^2 w}{\partial t^2} - w'v' + \frac{2}{r^2} wv = \int_{\Omega} \left(- \frac{1+\nu}{1-\nu} \beta \frac{\partial T}{\partial r} \right) v \qquad \textbf{(67)}$$

Eqs. (66) and (67) are both well-posed variational statements suitable for a finite element implementation. The only questionable term is the zero order coefficient in Eq. (67), where a singularity arises at $r = 0$. Fortunately, as stated before, the substitution of Eq. (39) when carried into the boundary conditions creates a Dirichlet condition at $r = 0$.

The net effect of this is to make integral evaluations at that point unnecessary. This will be shown shortly.

4.2.2 Stiffness, Mass and Load Integrals

Expressions for the element contributions to the global stiffness and mass matrices and load vector follow directly from the variational forms of the equations. They are presented here first for Eq. (66) and then for Eq. (67):

$$k_{ij}^e = \int_{\Omega_e} \psi'^e_i \, \psi'^e_j + \left(\frac{\Sigma_a - \nu \Sigma_f}{D}\right) \psi_i^e \psi_j^e \, d\Omega \tag{68}$$

$$m_{ij}^e = \int_{\Omega_e} \left(\frac{1}{V D}\right) \psi_i^e \psi_j^e \, d\Omega \tag{69}$$

$$f_i^e = \int_{\Omega_e} (S_{ext}) \psi_i^e \, d\Omega \tag{70}$$

Similar expressions hold for the thermoelasticity equation:

$$k_{ij}^e = \int_{\Omega_e} \psi'^e_i \, \psi'^e_j + \left(\frac{2}{r^2}\right) \psi_i^e \psi_j^e \, d\Omega \tag{71}$$

$$m_{ij}^e = \int_{\Omega_e} \left(\frac{1}{c^2}\right) \psi_i^e \psi_j^e \, d\Omega \tag{72}$$

$$f_i^e = \int_{\Omega_e} \left(-\frac{1+\nu}{1-\nu} \beta \frac{\partial T}{\partial r}\right) \psi_i^e \, d\Omega \tag{73}$$

4.2.3 Master Element

The selection of a master element is not as necessary in a one-dimensional code where a formal coordinate transformation is not implemented. However, since every

element is the same type (*i.e.*, there is no refinement of the order of the shape functions), it is useful to define a master element for purposes of clarity.

A linear element was selected for *ODMain*. While higher order polynomials would provide better interpolation over each element, they increase the number of degrees of freedom in the problem. Also, in the code implementation, all the material and load data spatial dependence is done on a piecewise constant element-by-element basis. This means refining the mesh gives better resolution of these spatially dependent data. Finally, implementing the implicit time-marching scheme requires the solution of a number of equations directly dependent on the number of degrees of freedom. It behooves us to minimize that number over each time step. Figure 4.1 is an illustration of a linear master element in one dimension:

Figure 4.1: Linear master element in one dimension

The local degrees of freedom are numbered right to left, and the master element dimension goes from –1 to 1.

4.2.4 Shape Functions

The shape functions defined on the master element in Figure 4.1 are as follows for a linear element:

$$\hat{\psi}_1(\xi) = \frac{1}{2}(1-\xi)$$
$$\hat{\psi}_2(\xi) = \frac{1}{2}(1+\xi) \quad \textbf{(74)}$$

The derivative for ψ_1 is simply –0.5, and for ψ_2 it is 0.5. In terms of a mesh coordinate *r* the expressions are element-dependent. These expressions are implemented in the code.

4.2.5 Finite Element Mesh Structure and Refinement

The finite element mesh in a linear one-dimensional code is necessarily simple. Figure 2 illustrates the basic structure of the mesh:

Figure 4.2: Mesh structure for 1D code

Figure 4.2 shows the finite element mesh for the one-dimensional code if only two elements were employed. Element 1 covers the solid sphere region, and element 2 the air or vacuum region required by the diffusion equation vacuum boundary condition.

The code obviously has an option to refine the mesh. All refinement takes place in the solid region, leaving one element outside for diffusion. The elasticity equations are only solved in the solid region, since displacement and density changes in the air element

do not change the neutron behavior noticeably. The air element always remains a distance *d* from the solid surface, as is appropriate for the diffusion equation boundary condition.

4.2.6 Boundary Conditions

The boundary conditions of the classical problem statement change when the substitution of Eq. (39) is employed. We illustrate how this occurs in the following derivation:

$$\begin{aligned} &\frac{\partial \phi}{\partial r} = \frac{1}{r}\frac{\partial w}{\partial r} - \frac{1}{2r^2}w \\ &\frac{1}{r}\frac{\partial w}{\partial r} - \frac{1}{2r^2}w = 0 \\ &2r\frac{\partial w}{\partial r} - w = 0 \\ &r = 0 \\ &-w = 0 \end{aligned} \tag{75}$$

This illustrates how pure Neumann conditions change to pure Dirichlet conditions under the transformation in Eq. (39).

The conditions for diffusion are very simply that the transformed field variable *w* vanishes at the boundaries of the problem domain. The size of the air region outside the sphere is determined by neutron transport theory. To a close approximation, it is 2.13*D*, where *D* is the diffusion length (Lamarsh, 2001).

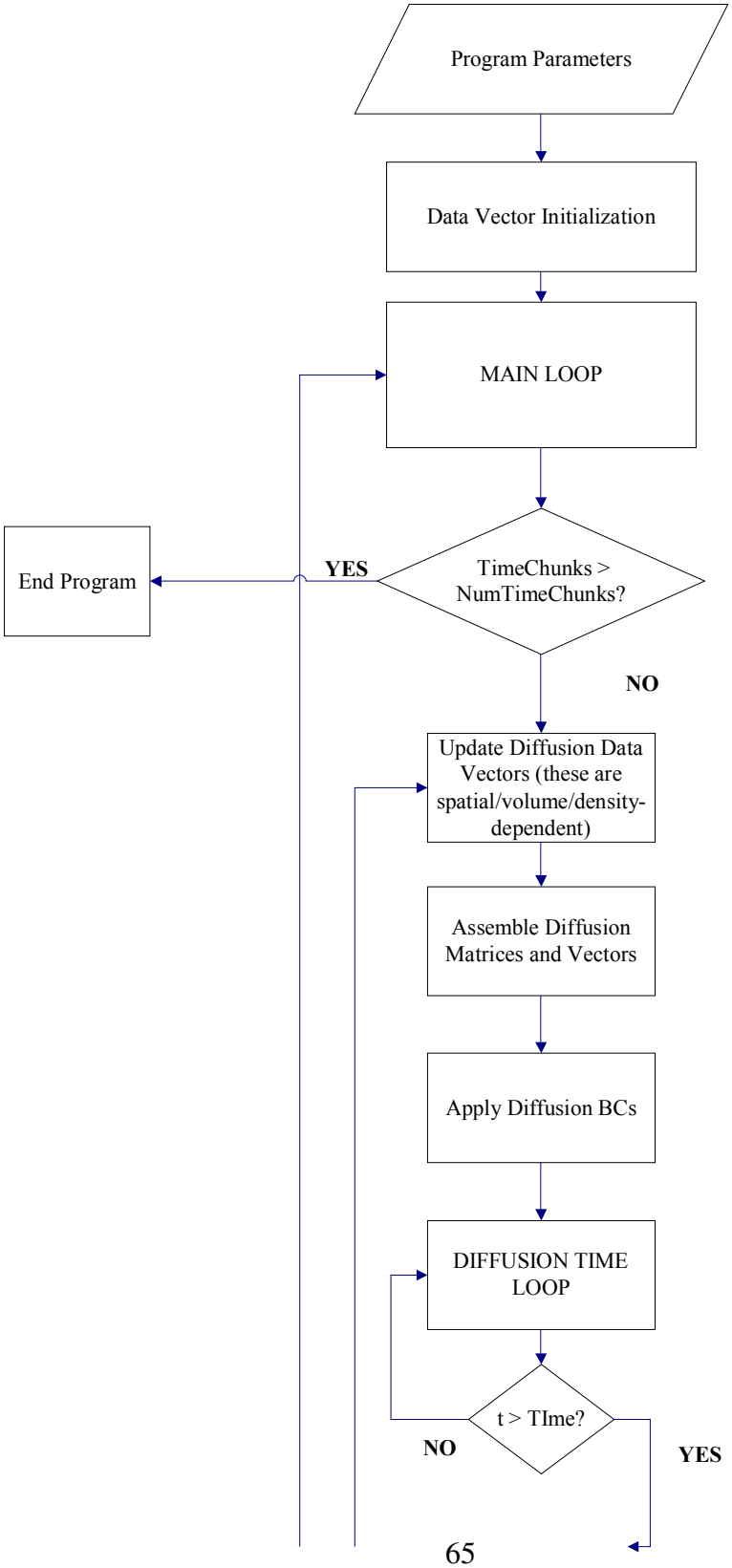
For elasticity, the conditions are zero displacement at the sphere center and zero radial stress at the solid surface. This second condition is a general boundary condition as described in Chapter 3.

4.2.7 Other Details

The integration order for the mesh is a four-point Newton-Cotes rule, commonly known as Simpson's 3/8ths rule. We use Newton-Cotes because the shape functions employed are actually defined on the mesh coordinate system instead of the master element. This choice was made simply for ease of coding.

The user may change the time-marching scheme. It is by default a trapezoid rule for both the diffusion and elasticity equations. See Section 3.3.6 for details on the time-marching algorithm, as well as Appendices A and B for the actual code.

Figure 4.3 illustrates the general structure of both the one and two-dimensional codes. For more in-depth explanation, see Appendix B.


Program Parameters
Data Vector Initialization
MAIN LOOP
TimeChunks > NumTimeChunks?
YES
End Program
NO
Update Diffusion Data Vectors (these are spatial/volume/density-dependent)
Assemble Diffusion Matrices and Vectors
Apply Diffusion BCs
DIFFUSION TIME LOOP
t > TIme?
NO
YES

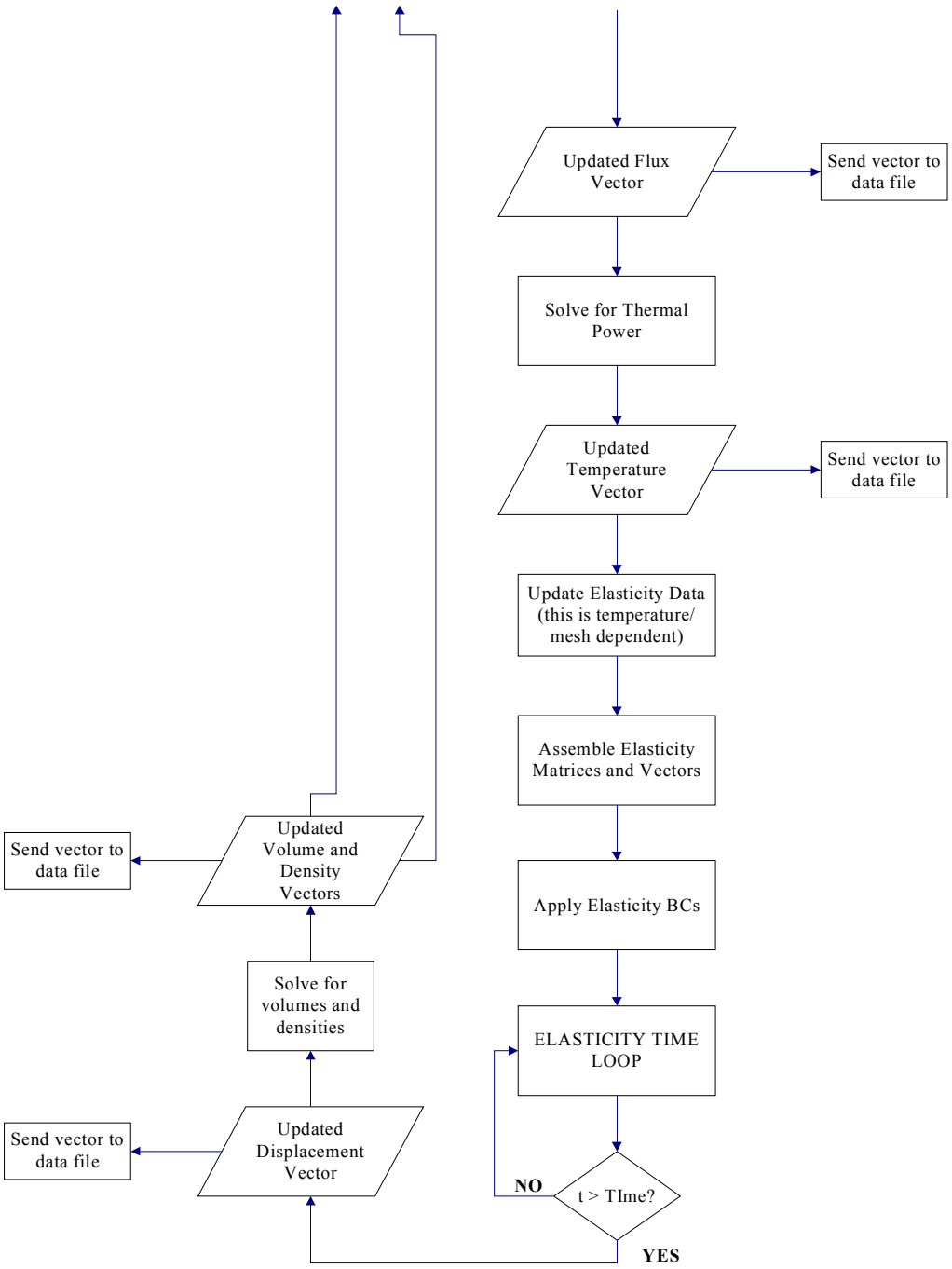


Figure 4.3: Flowchart of basic program logic in *ODMain* and *TDMain*

4.3 TWO-DIMENSIONAL CODE

The development of a code to simulate similar behaviors in two-dimensional axisymmetric coordinates is somewhat more complicated. However, we follow the same general steps.

4.3.1 Variational Forms

The variational form for diffusion appears very similar to that of the one-dimensional case. For elasticity, the variational form is acquired via the principal of virtual work (Fung and Tong, 2001). The integrals for element matrices and vectors are expressed in vector and matrix form, instead of component-wise as seen in Eqs. (68-73). First we give the variational form for two-dimensional diffusion on axisymmetric coordinates:

$$\int_{\Omega} \frac{1}{V D} \frac{\partial u}{\partial t} - u_{,r} v_{,r} - u_{,z} v_{,z} + \left(\frac{\Sigma_a - \nu \Sigma_f}{D}\right) uv = \int_{\Omega} v S_{ext} \qquad \textbf{(76)}$$

The variational form for plane and axisymmetric elasticity arises from the principle of virtual work. The static problem is one of minimizing a potential energy functional. In a dynamic problem, we must solve for a stationary Hamiltonian (Fung and Tong, 2001). The potential energy is first given as:

$$\Pi_e = \int_{A_e} \left(\frac{1}{2} e^T D_e e - u^T b\right) dA - \int_{\partial A_{\sigma e}} u^T \bar{T}\, ds \qquad \textbf{(77)}$$

where e is the strain vector, D is a coefficient matrix, u is the displacement vector, and T is the surface traction vector. This permits us to define stiffness and load integrals,

which will be given explicitly in the next section. The element stiffness matrix and load vector enter the Hamiltonian as:

$$H = \int_{t1}^{t2} \left\{ \sum_{e=1}^{N} \left[\frac{1}{2} q_e^T k_e q_e - q_e^T f_e - \frac{1}{2} \dot{q}_e^T m_e \dot{q}_e \right] + \bar{F}_1 q_1 - \bar{F}_2 q_{N+1} \right\} dt \tag{78}$$

Summing over elements gives:

$$H = \int_{t1}^{t2} \left(\frac{1}{2} u^T K u - u^T F - \frac{1}{2} \dot{u}^T M \dot{u} + \bar{F}_1 q_1 - \bar{F}_2 q_{N+1} \right) dt \tag{79}$$

where *K*, *F* and *M* are the stiffness, load and mass, respectively. Setting the first variation δH with respect to the nodal parameter *u* to zero, integrating by parts, and making use of $\delta u(t_1) = \delta u(t_2) = 0$, we have the familiar matrix equation seen in Eq. (57), with zero damping:

$$M \ddot{u} + Ku = F \tag{80}$$

4.3.2 Stiffness, Mass and Load Integrals

The stiffness, mass and load integrals for diffusion are defined here entry-by-entry (i.e., the form seen in Eqs. (68-70). The corresponding integrals for axisymmetric elasticity are given in matrix and vector form.

$$k_{ij}^e = \int_{\Omega_e} \left(\frac{\partial \psi_i^e}{\partial r} \frac{\partial \psi_j^e}{\partial r} + \frac{\partial \psi_i^e}{\partial z} \frac{\partial \psi_j^e}{\partial z} + \left(\frac{\Sigma_a - \nu \Sigma_f}{D} \right) \psi_i^e \psi_j^e \right) r dr dz \tag{81}$$

$$m_{ij}^e = \int_{\Omega_e} (\frac{1}{V\,D}) \psi_i^e \psi_j^e \, r dr dz \tag{82}$$

$$f_i^e = \int_{\Omega_e} (S_{ext}) \psi_i^e \, r dr dz \tag{83}$$

$$k = \int_{\Omega_e} (d_a h)^T D_a (d_a h) \, r dr dz \tag{84}$$

$$m = \int_{\Omega_e} (h)^T \rho(h) \, r dr dz \tag{85}$$

$$f = \int_{\Omega_e} (h)^T b \, r dr dz + \int_{\partial\Omega_e} (h)^T \bar{T} \, ds \tag{86}$$

The terms in Eqs. (81-83) should be familiar from the one-dimensional case. In Eqs. (84-86), the terms are defined as follows for an axisymmetric formulation:

$$d_a^T = \begin{bmatrix} \frac{\partial}{\partial r} & \frac{1}{r} & 0 & \frac{\partial}{\partial z} \\ 0 & 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial r} \end{bmatrix} \tag{87}$$

$$h = \begin{bmatrix} \psi_1 & 0 & \psi_2 & 0 & \psi_3 & 0 & \psi_4 & 0 \\ 0 & \psi_1 & 0 & \psi_2 & 0 & \psi_3 & 0 & \psi_4 \end{bmatrix} \tag{88}$$

$$D = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 \\ \nu & 1-\nu & \nu & 0 \\ \nu & \nu & 1-\nu & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \tag{89}$$

Other terms in Eqs. (84-86) are the density ρ, the body force vector *b*, and the surface traction vector $\bar{T}$. Note that all integrations over elements carry an additional *r* factor.

4.3.3 Master Element

The choice of master element in the two-dimensional case is much more important than in one dimension. Since the domain in the axisymmetric formulation is rectangular in cross-section, selecting a quadrilateral master element seems logical. The order of the element is once again linear, for much the same considerations given in the one-dimensional case. Figure 4.4 shows the master quadrilateral element used in *TDMain*:

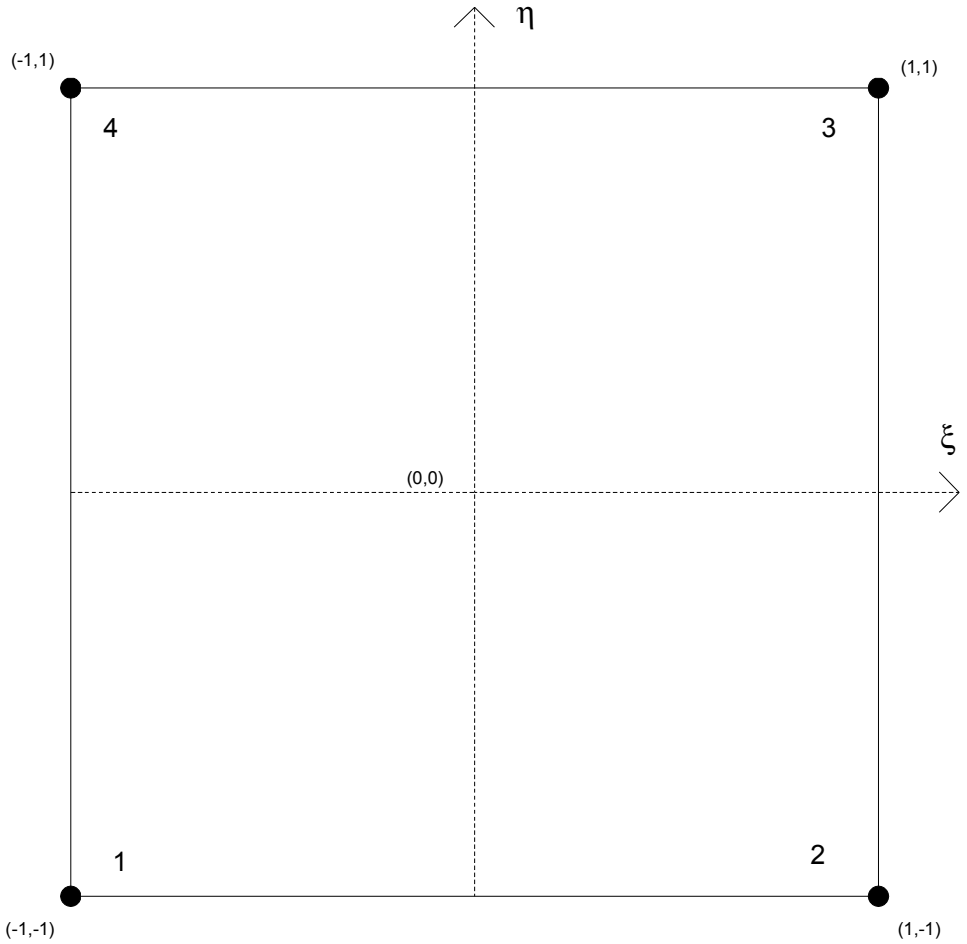


Figure 4.4: Linear quadrilateral master element

Figure 4.4 gives the local degree of freedom at each node, as well as its coordinates in the (ξ,η) system. The local degree of freedom number corresponds to the shape function number at that node.

4.3.4 Shape Functions

The shape functions defined on the master element are given here:

$$\hat{\psi}_1 = \frac{1}{4}(1-\xi)(1-\eta)$$
$$\hat{\psi}_2 = \frac{1}{4}(1+\xi)(1-\eta) \quad \textbf{(90)}$$
$$\hat{\psi}_3 = \frac{1}{4}(1+\xi)(1+\eta)$$
$$\hat{\psi}_4 = \frac{1}{4}(1-\xi)(1+\eta)$$

Global derivatives used in evaluating integrals are obtained via the chain rule and the definitions in Eq. (90):

$$\frac{\partial \psi_j^e}{\partial r} = \frac{\partial \hat{\psi}_j}{\partial \xi}\frac{\partial \xi}{\partial r} + \frac{\partial \hat{\psi}_j}{\partial \eta}\frac{\partial \eta}{\partial r} \quad \textbf{(91)}$$
$$\frac{\partial \psi_j^e}{\partial z} = \frac{\partial \hat{\psi}_j}{\partial \xi}\frac{\partial \xi}{\partial z} + \frac{\partial \hat{\psi}_j}{\partial \eta}\frac{\partial \eta}{\partial z}$$

The terms defined in Eq. (90) and Eq. (91) can be seen directly in Eqs. (81-83). They enter into the elasticity formulation via the definitions in Eq. (87) and Eq. (88).

4.3.5 Finite Element Mesh Structure and Refinement

The finite element mesh in the two-dimensional code must also account for diffusion boundary conditions. Further, instead of a solid object as modeled in the one-dimensional code, the two-dimensional code accounts for a hollow cavity in the cylinder. Figure 4.5 shows the general form of the mesh employed in the two-dimensional code:

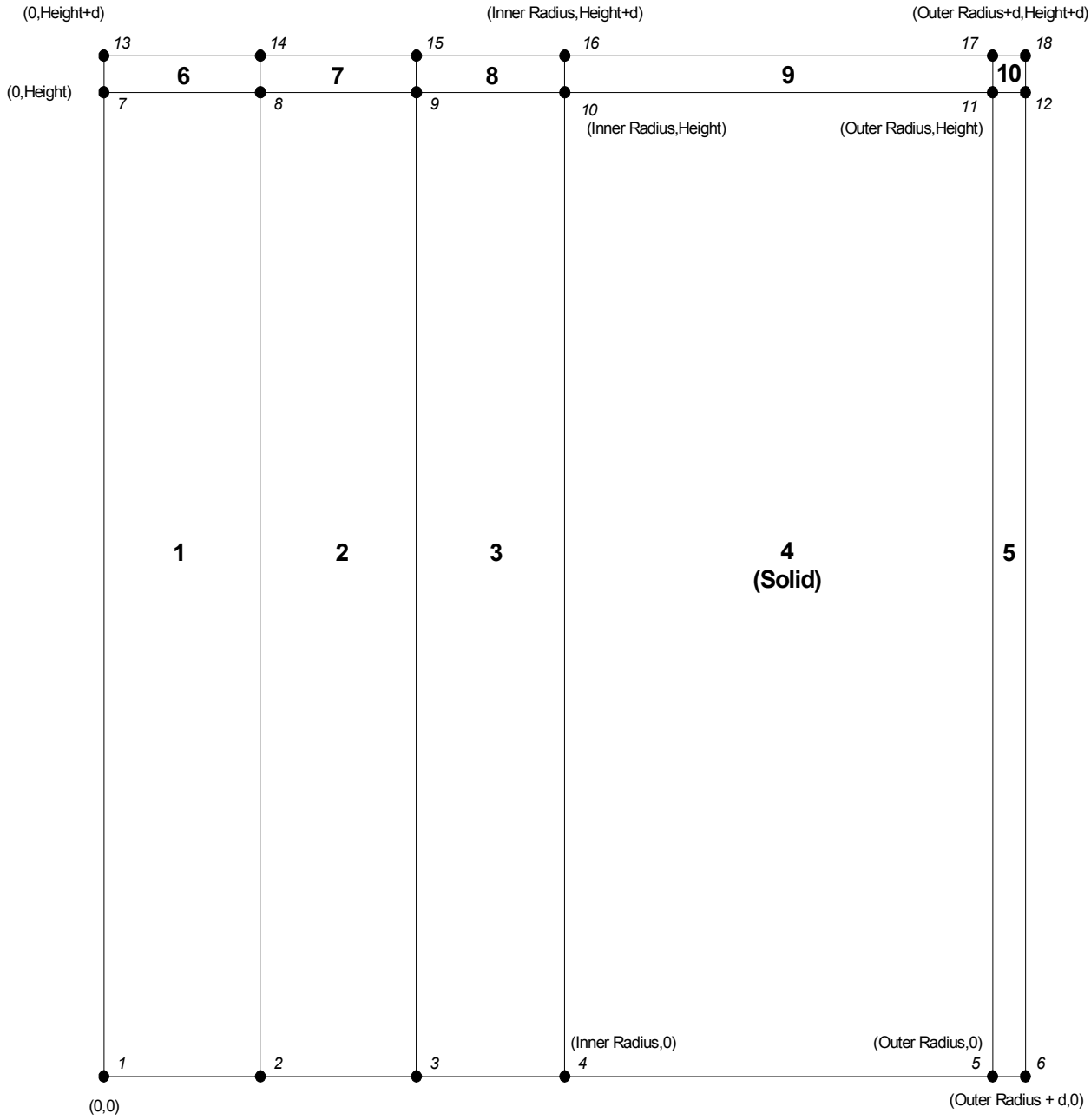


Figure 4.5: Mesh structure for two-dimensional code

Figure 4.5 shows the coarsest possible mesh for the two-dimensional code. Every element except element 4 is air or vacuum, as specified by the user. All mesh refinement takes place in the solid region, which can be split into however many elements the user desires. The elasticity equations are solved only in the solid region.

4.3.6 Boundary Conditions

The boundary conditions for the diffusion solution are much the same as those encountered in the one-dimensional solver. At $r = 0$ and $z = 0$, there are pure Neumann conditions requiring that the spatial derivative goes to zero at those planes. At the other problem boundaries (the top and right boundaries seen in Figure 4.5), there are pure Dirichlet conditions requiring the flux to go zero.

The formulation of the elasticity equations in sections 4.3.1 and 4.3.2 make implementing the appropriate boundary conditions especially simple. First, note that the elasticity solution has two degrees of freedom at each node: one for r displacement and one for z displacement. The boundary conditions are as follows:

- z-displacement at $z = 0$ equals zero (Dirichlet)
- z-traction at $z =$ *Height* is as specified in section 3.2.4
- r-traction at $r =$ *Inner Radius* is as specified in section 3.2.4
- r-traction at $r =$ *Outer Radius* is as specified in section 3.2.4

The first boundary condition is the familiar pure Dirichlet type. The other three boundary conditions simply specify tractions ($\bar{T}$ in Eq. (86)) according to the Duhamel-

Neumann analogy. These are known as traction boundary conditions and are common in multi-dimensional elasticity.

4.3.7 Other Details

The integration rule for the mesh is a two-point Gauss rule. This is the standard integration rule for linear quadrilateral elements (Hughes, 2001). It integrates the linear shape functions exactly.

As in the one-dimensional code, the user may change the time-marching scheme. It is by default a trapezoid rule for both the diffusion and elasticity equations. See Section 3.3.6 for details on the time-marching algorithm, as well as Appendices C and D for the code.

Again, Figure 4.3 illustrates the general structure of the code. For a more in-depth explanation of parameters, modules and procedures, see Appendix D.

Chapter 5: Other Computational Methods

5.1 Numerical Solution of Dynamic Thermomechanical Equation

Prior to development of the finite element codes in this dissertation, a method-of-lines solution of the approximate one-dimensional radial displacement equation seen in Eq. (35) coupled to a Monte Carlo solution of the transport equation was used to attempt to reproduce SPR III operational results. This permitted taking neutron reflection into consideration (functionality beyond the neutron diffusion approximation employed in the finite element implementation). While this method required a prohibitive amount of time and user interaction, it produced accurate results. The general outline of this method is presented here, while pertinent results are included in Chapter 6.

The applicable thermoelastic equation is a second order hyperbolic partial differential equation. Equations of this type typically do not have closed form solutions, as is the case here. This means a numerical solution is required.

The numerical method-of-lines implementation in Mathsoft's © Mathcad 11 software was chosen for ease of use. While using an interface like Mathcad necessarily increases computing time, the time saved in development, especially with regard to temperature-dependent coefficients and initial conditions, was deemed worthwhile. The Nordheim-Fuchs kinetics equations were also solved in Mathcad. A flow chart of the numerical calculation is most effective in explaining the procedure.

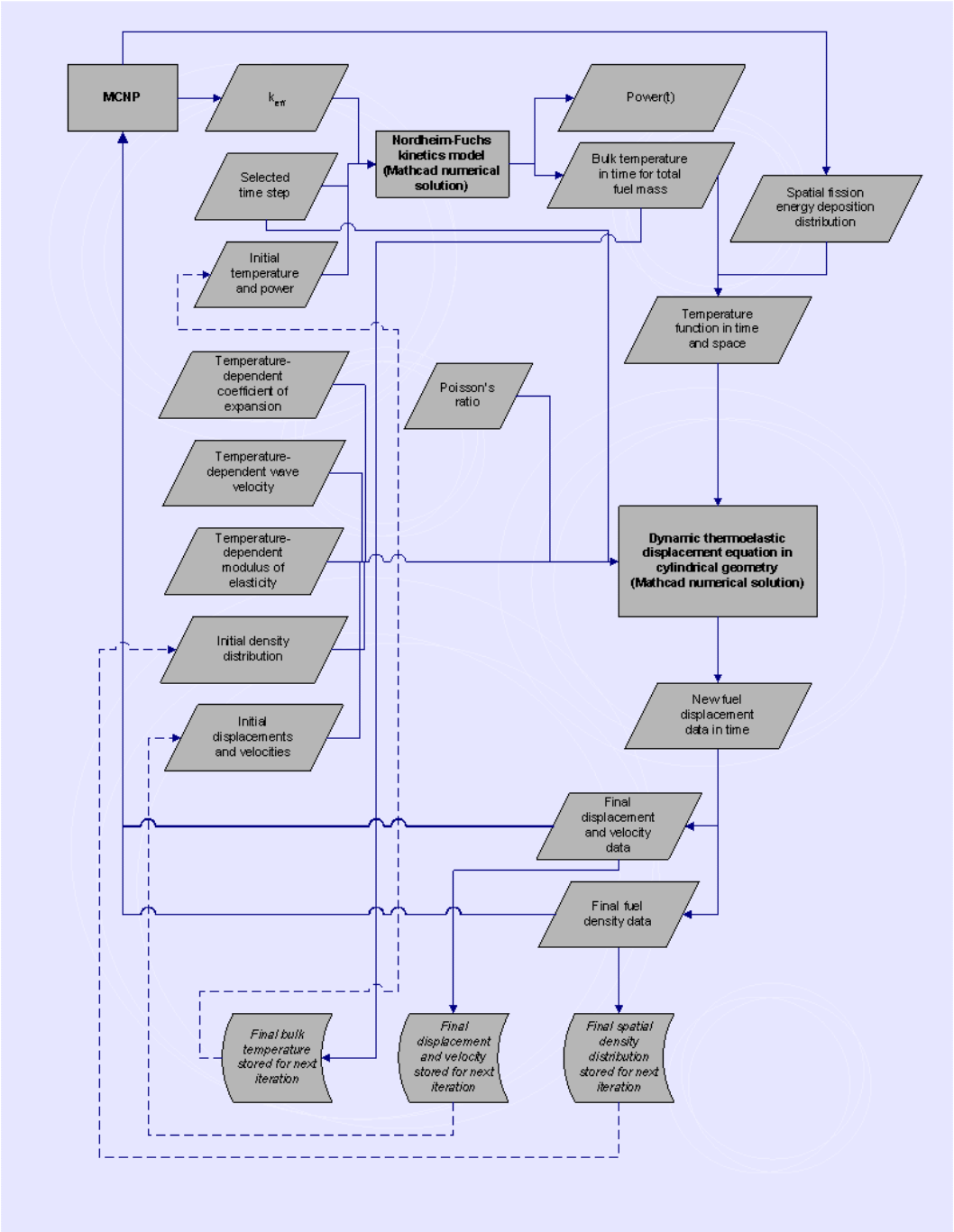


Figure 5.1: Flow chart of method-of-lines/MCNP numerical calculation

By observing the flow chart, one can see that almost all the numerical steps take place within Mathcad. MCNP produces only the k_{eff} and spatial energy deposition data.

Seventeen iterations of this process were used to produce the "small" pulse presented in Chapter 6, and nineteen were used for the "large" pulse.

5.2 MCNP

MCNP, or Monte Carlo N-Particle transport code, is a general-purpose, continuous-energy, generalized-geometry, and time-dependent coupled neutron/photon/electron Monte Carlo transport code (Briesmeister 2000). The user creates an input deck containing the following information:

- geometry specification,
- description of materials and selection of cross-section evaluations,
- location and characteristics of the neutron, photon, or electron source,
- type of answers or tallies desired;

Our implementation of MCNP takes advantage of only two of its capabilities: k_{eff} eigenvalue calculation for a fissile system, and fission energy tallying.

MCNP theoretically reproduces statistical processes (such as nuclear interactions). Statistical codes of this nature are useful when solving a problem on a complicated domain. While solving the neutron transport equation via the method of lines and coupling it directly to the dynamic thermoelastic equations represents the mathematically "pure" solution route, solving the transport equation using a deterministic versus statistical method sometimes creates more problems than it solves (i.e., complicated and numerous boundary and initial conditions, and multiple interface conditions between many types of materials). Even though MCNP cannot change its

geometry during a single calculation (*i.e.*, it solves the neutron transport equation in time, not any mechanics equations), obtaining many static slices and modifying the input geometry and material density each time makes a "time-dependent" solution possible.

The time step size chosen for each iteration was based on the average temperature rise given by the kinetics model. Exceeding a rise of ~40 K over an iteration tends to produce poor results. Further refinement over the peak of the power profile was also required.

Chapter 6: Results and Discussion for Method-of-Lines/MCNP Numerical Method

6.1 SMALL PROMPT INSERTION

The results of the method-of-lines/MCNP numerical simulations are divided into the thermomechanical and neutron transport aspects of two complete SPR III pulses. The first pulse is produced by an addition of ~$1.0846 via the burst element (i.e., the top of the burst element is 14 cm above the core midline, see Chapter 2 for clear illustrations of reactor configuration and burst element positioning). A prompt addition of ~8.46 cents falls in the low-to-middle range of the SPR III operational envelope (see Table 2.2). The fuel mass is divided for thermomechanical treatment into three sections: the upper eight plates, the middle two plates, and the lower eight plates (see Figure 2.9). First we shall present the radial displacement results in time for all three fuel sections, then the k_{eff}, power, and temperature curves resulting from MCNP-based iterative results deriving from those displacements.

6.1.1 Thermoelastic Displacement Results for $1.0846 pulse

The radial displacement of SPR III during a pulse induced by a moderate prompt reactivity insertion is described by the following three figures.

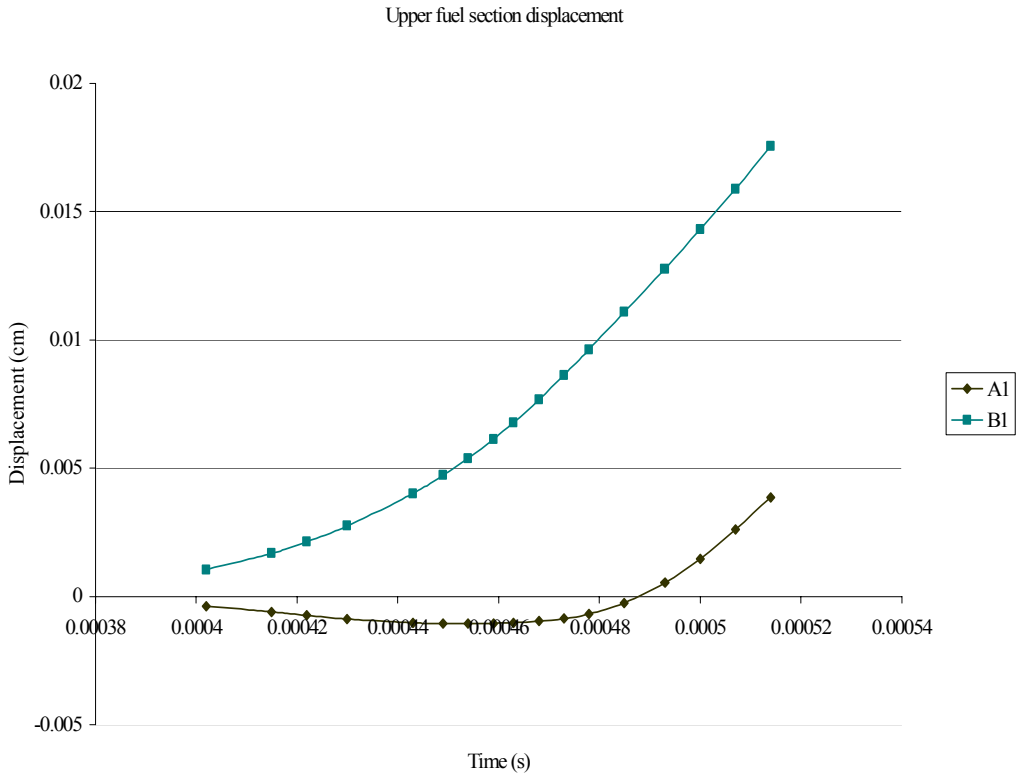


Figure 6.1: Upper fuel section radial displacement for $1.0846 pulse

In this figure, as in those that follow shortly, the "A" curve represents the displacement of the interior surface of the annulus, and the "B" curve represents the displacement of the exterior surface. Next we present the displacement for the middle and lower fuel sections.

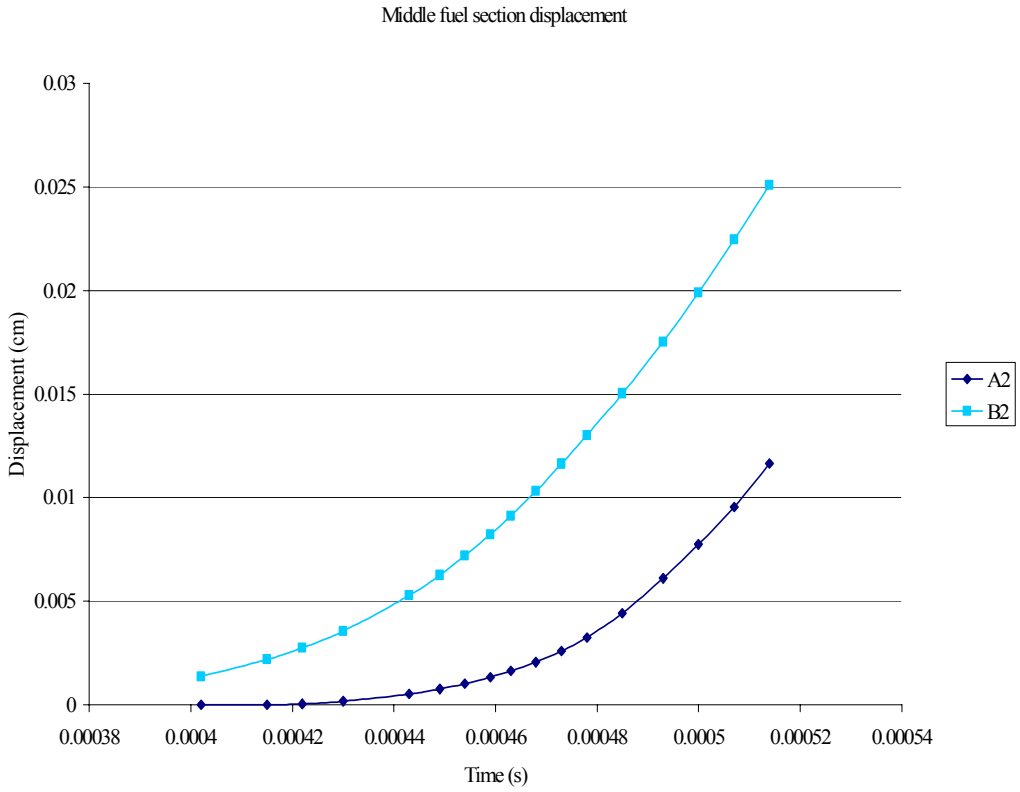


Figure 6.2: Middle fuel section radial displacement for $1.0846 pulse

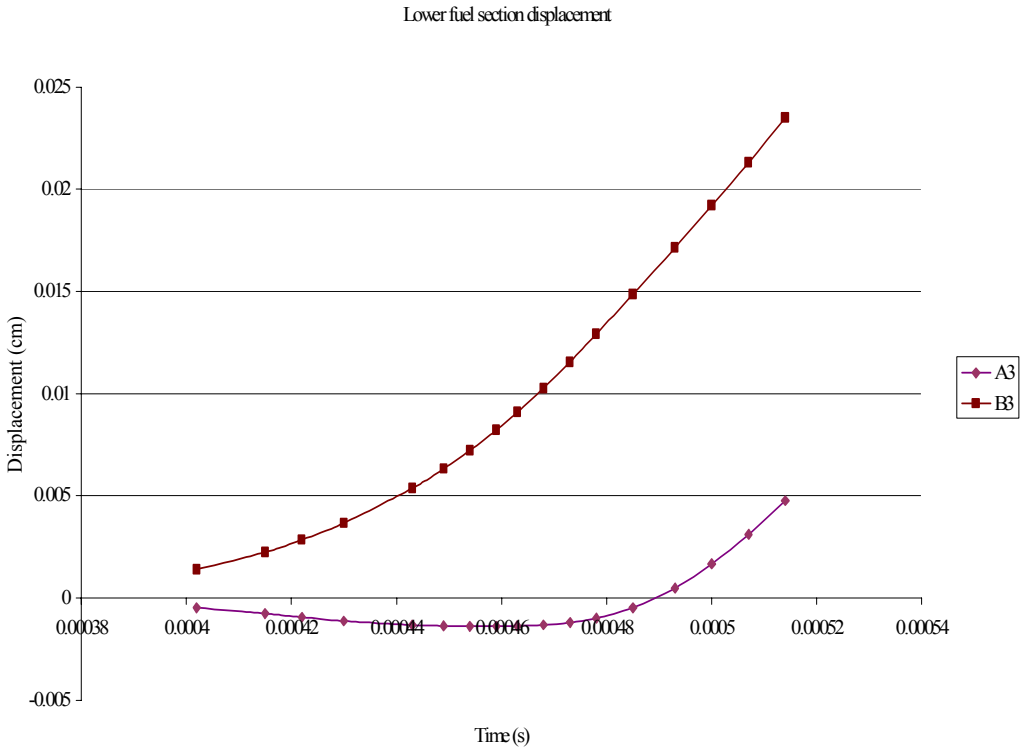


Figure 6.3: Lower fuel section radial displacement for $1.0846 pulse

Table 6.1 summarizes the data illustrated in these plots.

Time (s)	A1 (cm)	B1 (cm)	A2 (cm)	B2 (cm)	A3 (cm)	B3 (cm)
0.000000	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
0.000402	-3.76E-04	1.05E-03	1.21E-06	1.37E-03	-4.75E-04	1.40E-03
0.000415	-5.99E-04	1.68E-03	5.95E-06	2.19E-03	-7.61E-04	2.24E-03
0.000422	-7.38E-04	2.13E-03	4.93E-05	2.76E-03	-9.39E-04	2.85E-03
0.000430	-8.80E-04	2.75E-03	1.81E-04	3.56E-03	-1.13E-03	3.67E-03
0.000443	-1.03E-03	4.01E-03	5.26E-04	5.28E-03	-1.33E-03	5.37E-03
0.000449	-1.05E-03	4.73E-03	7.61E-04	6.26E-03	-1.37E-03	6.33E-03
0.000454	-1.06E-03	5.39E-03	1.01E-03	7.20E-03	-1.39E-03	7.22E-03
0.000459	-1.05E-03	6.13E-03	1.34E-03	8.23E-03	-1.39E-03	8.22E-03
0.000463	-1.02E-03	6.78E-03	1.63E-03	9.12E-03	-1.37E-03	9.09E-03
0.000468	-9.65E-04	7.67E-03	2.07E-03	1.03E-02	-1.31E-03	1.03E-02
0.000473	-8.60E-04	8.62E-03	2.60E-03	1.16E-02	-1.20E-03	1.16E-02
0.000478	-6.78E-04	9.63E-03	3.26E-03	1.30E-02	-9.88E-04	1.29E-02
0.000485	-2.46E-04	1.11E-02	4.44E-03	1.50E-02	-4.71E-04	1.49E-02
0.000493	5.32E-04	1.28E-02	6.11E-03	1.75E-02	4.88E-04	1.71E-02
0.000500	1.47E-03	1.43E-02	7.75E-03	1.99E-02	1.67E-03	1.92E-02
0.000507	2.60E-03	1.59E-02	9.56E-03	2.25E-02	3.12E-03	2.13E-02
0.000514	3.87E-03	1.75E-02	1.17E-02	2.51E-02	4.76E-03	2.35E-02

Table 6.1: Summary of radial displacement data for $1.0846 pulse

Note in Table 6.1 the column labels ending in “1” refer to the upper fuel section, “2” refer to the middle, and “3” refer to the lower.

Note also that displacement data was generated for twenty concentric points in each upper and lower annulus, and twelve in the smaller middle sections. These data were used in updating the MCNP deck geometry from iteration to iteration. This displacement data is used in MCNP, but not presented here because all other displacements fall between the extremes of the outer and inner surfaces.

6.1.2 Neutron Kinetics Results for $1.0846 pulse

Now we present the total fission power, bulk temperature, and k_{eff} curves for a $1.0846 pulse. Each figure illustrates the comparison of the results to the corresponding symmetric Nordheim-Fuchs profile.

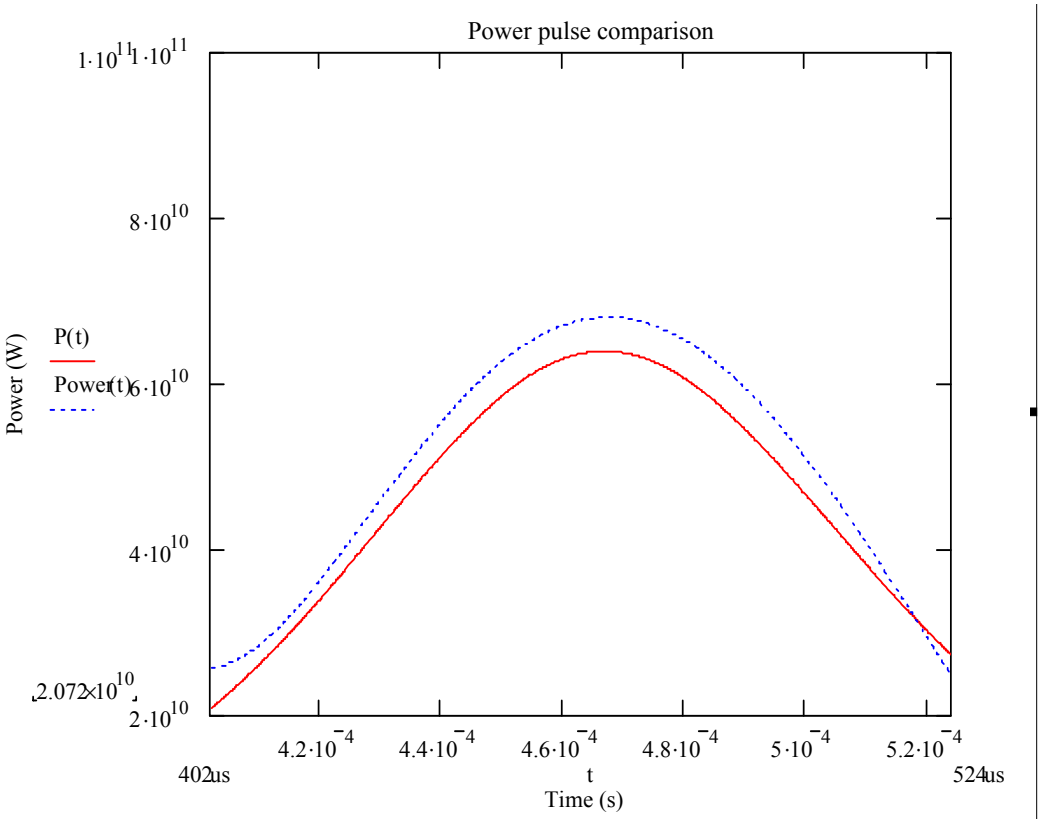


Figure 6.4: Power profile comparison for $1.0846 pulse

In Figure 6.4 and those following, the dotted line represents a polynomial fit of the power data produced using the method-of-lines/MCNP method, and the solid line represents a symmetric Nordheim-Fuchs fit of operational data seen in Table 2.2. Full-width-half-maximum (FWHM) for the theoretical pulse fit to SPR III data is ~100.1 μs, and ~98.49 μs for the method-of-lines/MCNP pulse.

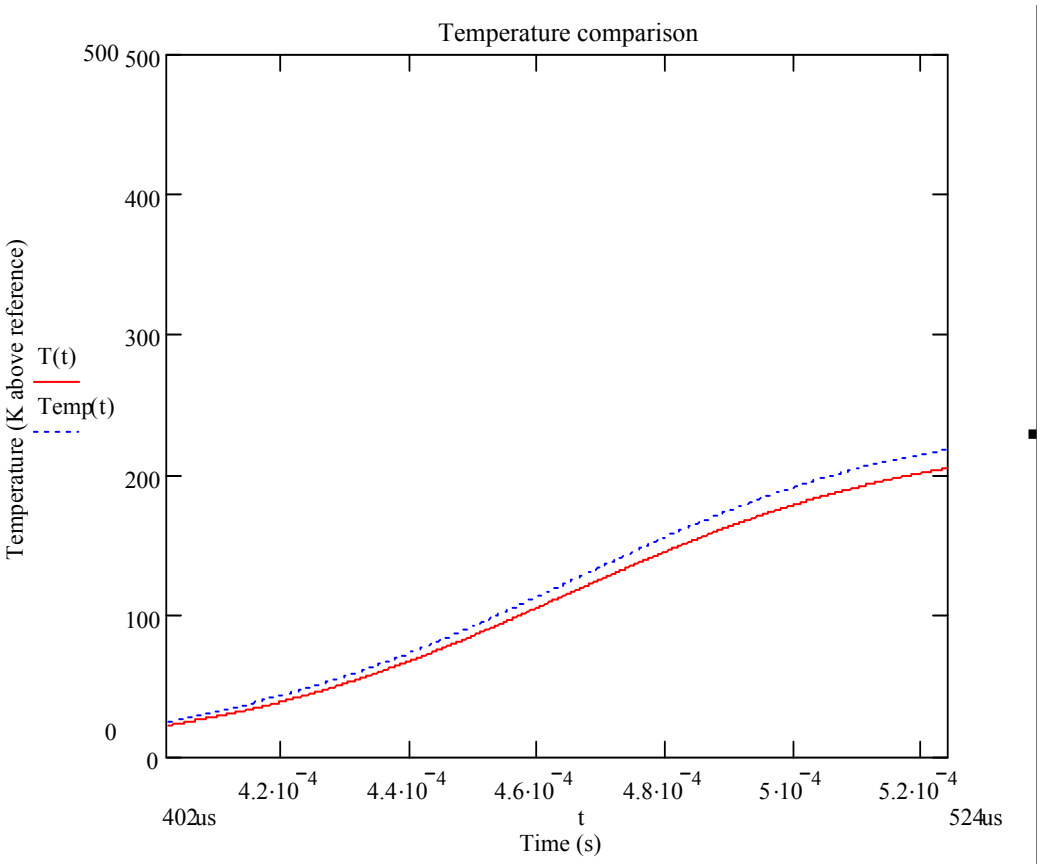


Figure 6.5: Temperature profile comparison for $1.0846 pulse

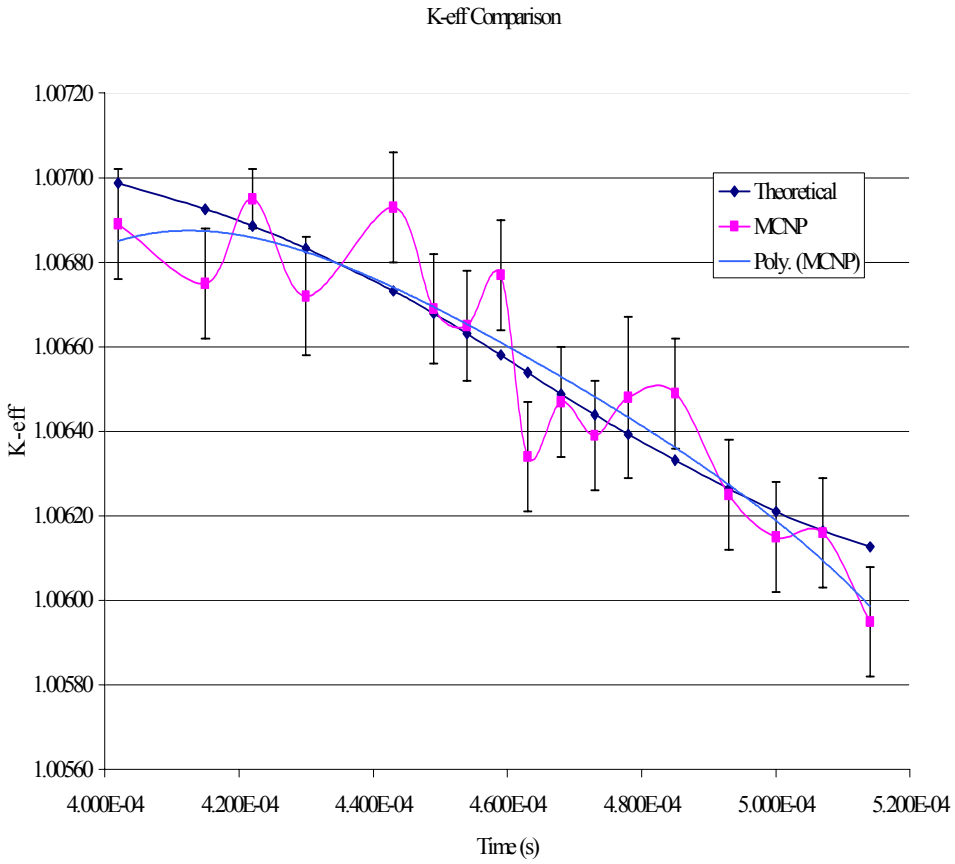


Figure 6.6: K-eff comparison for $1.0846 pulse

Table 6.2 summarizes the information plotted above.

Time (s)	Power (W)	delta T (K)	MCNP k_{eff}	MCNP stdev
0.000E+00	1.00000E+04	0.00	1.0071	0.0001
4.020E-04	2.51689E+10	24.39	1.00689	0.00013
4.150E-04	3.38775E+10	37.65	1.00675	0.00013
4.220E-04	3.72740E+10	46.21	1.00695	0.00007
4.300E-04	4.61877E+10	57.54	1.00672	0.00014
4.430E-04	5.37586E+10	79.25	1.00693	0.00013
4.490E-04	6.26414E+10	90.72	1.00669	0.00013
4.540E-04	6.57553E+10	101.17	1.00665	0.00013
4.590E-04	6.81217E+10	111.96	1.00677	0.00013
4.630E-04	7.23240E+10	120.92	1.00634	0.00013
4.680E-04	6.76609E+10	131.97	1.00647	0.00013
4.730E-04	6.60654E+10	142.43	1.00639	0.00013
4.780E-04	6.28313E+10	152.41	1.00648	0.00019
4.850E-04	6.10474E+10	165.69	1.00649	0.00013
4.930E-04	5.93825E+10	180.26	1.00625	0.00013
5.000E-04	5.18947E+10	191.88	1.00615	0.00013
5.070E-04	4.33081E+10	201.71	1.00616	0.00013
5.140E-04	3.63093E+10	209.87	1.00595	0.00013

Table 6.2: Summary of numerical kinetics results for $1.0846 pulse

These results, and those that follow for the large insertion, shall be summarized and explained at the end of this chapter.

6.2 LARGE PROMPT INSERTION

This section will mirror the previous section, except it refers to a pulse resulting from a $1.136 reactivity addition (*i.e.*, the top of the burst element is moved to 18.3 cm above core midline), representing the extreme upper operational range for SPR III. Again, radial displacement in time shall be presented for each fuel section, followed by neutron kinetics results.

6.2.1 Thermoelastic Displacement Results for $1.136 pulse

The thermoelastic radial displacements in time for each reactor fuel section during a method-of-lines/MCNP SPR III pulse resulting from ~13.6 cents of prompt reactivity are summarized in the following figures.

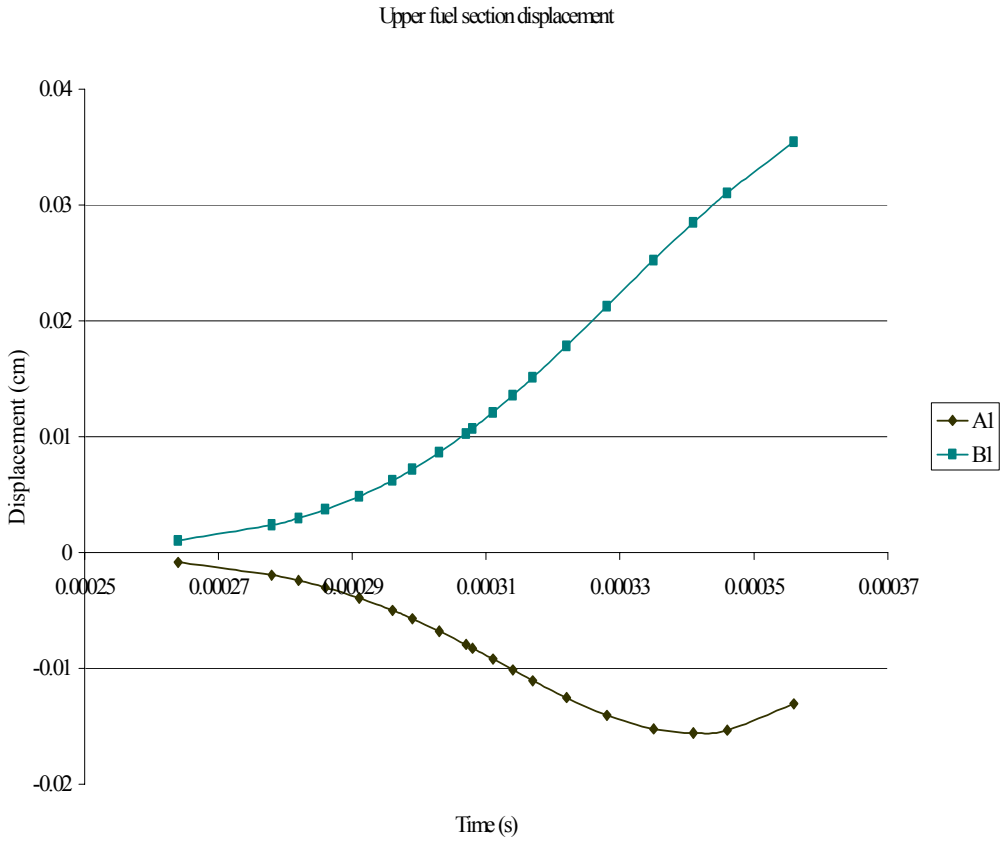


Figure 6.7: Upper fuel section radial displacement for $1.136 pulse

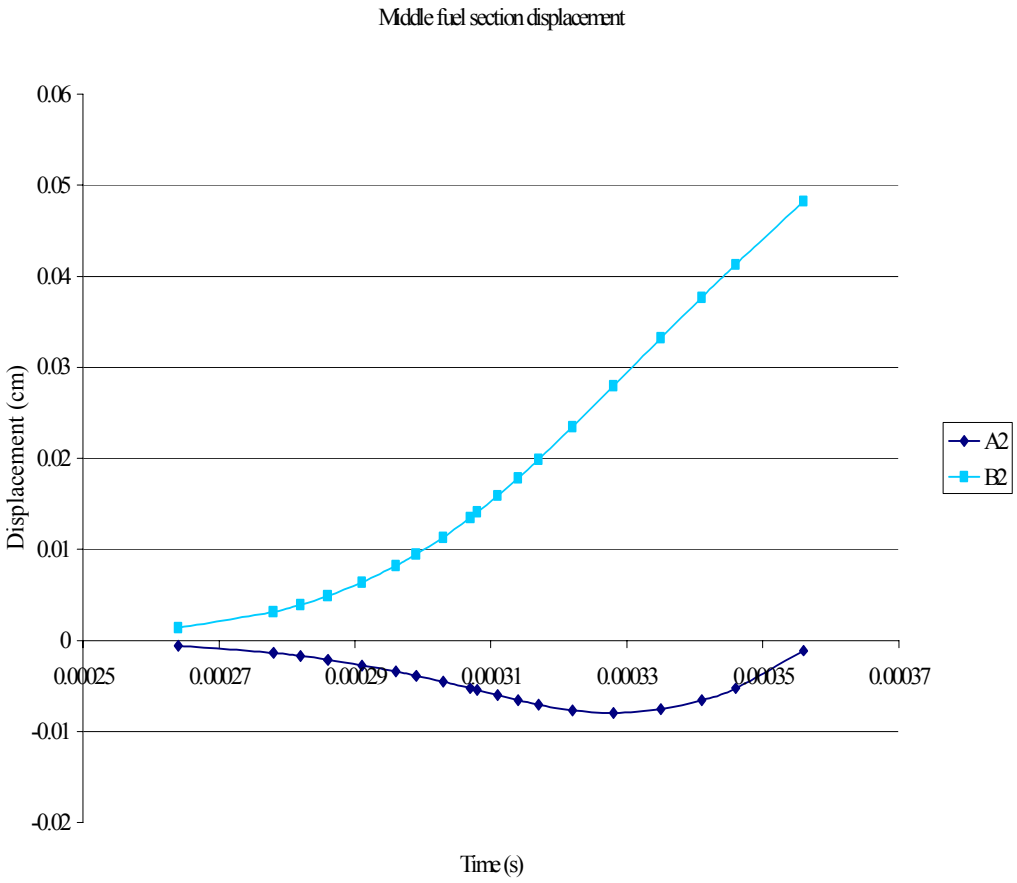


Figure 6.8: Middle fuel section radial displacement for $1.136 pulse

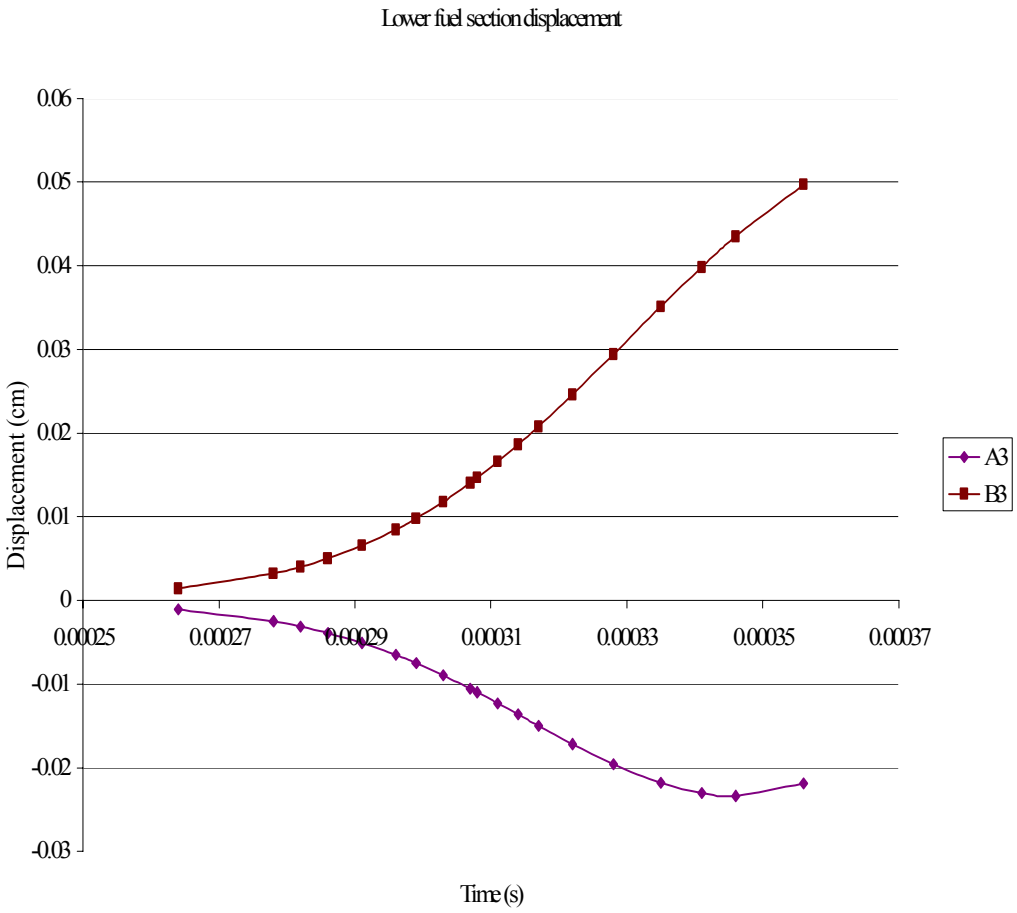


Figure 6.9: Lower fuel section radial displacement for $1.136 pulse

Table 6.3 summarizes the data plotted in the figures above.

Time (s)	A1 (cm)	B1 (cm)	A2 (cm)	B2 (cm)	A3 (cm)	B3 (cm)
0.000000	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
0.000264	-8.46E-04	1.05E-03	-5.88E-04	1.38E-03	-1.09E-03	1.42E-03
0.000278	-1.93E-03	2.38E-03	-1.36E-03	3.12E-03	-2.48E-03	3.21E-03
0.000282	-2.42E-03	2.98E-03	-1.71E-03	3.92E-03	-3.12E-03	4.03E-03
0.000286	-3.02E-03	3.72E-03	-2.13E-03	4.88E-03	-3.90E-03	5.03E-03
0.000291	-3.91E-03	4.84E-03	-2.74E-03	6.37E-03	-5.08E-03	6.57E-03
0.000296	-4.98E-03	6.22E-03	-3.42E-03	8.17E-03	-6.50E-03	8.46E-03
0.000299	-5.71E-03	7.19E-03	-3.88E-03	9.43E-03	-7.49E-03	9.78E-03
0.000303	-6.78E-03	8.64E-03	-4.54E-03	1.13E-02	-8.95E-03	1.18E-02
0.000307	-7.95E-03	1.03E-02	-5.26E-03	1.35E-02	-1.06E-02	1.41E-02
0.000308	-8.26E-03	1.07E-02	-5.47E-03	1.41E-02	-1.10E-02	1.47E-02
0.000311	-9.18E-03	1.21E-02	-6.01E-03	1.59E-02	-1.23E-02	1.66E-02
0.000314	-1.01E-02	1.36E-02	-6.56E-03	1.78E-02	-1.36E-02	1.86E-02
0.000317	-1.11E-02	1.51E-02	-7.07E-03	1.98E-02	-1.50E-02	2.08E-02
0.000322	-1.25E-02	1.78E-02	-7.69E-03	2.34E-02	-1.72E-02	2.46E-02
0.000328	-1.40E-02	2.13E-02	-7.98E-03	2.79E-02	-1.96E-02	2.94E-02
0.000335	-1.52E-02	2.52E-02	-7.56E-03	3.32E-02	-2.18E-02	3.51E-02
0.000341	-1.56E-02	2.85E-02	-6.58E-03	3.76E-02	-2.30E-02	3.98E-02
0.000346	-1.53E-02	3.11E-02	-5.27E-03	4.13E-02	-2.34E-02	4.35E-02
0.000356	-1.31E-02	3.55E-02	-1.12E-03	4.82E-02	-2.19E-02	4.97E-02

Table 6.3: Summary of radial displacement data for $1.136 pulse

6.2.2 Neutron Kinetics Results for $1.136 pulse

Next we present the same comparison plots for a $1.136 pulse. The fuel temperature coefficient of reactivity used for the Nordheim-Fuchs fit was -$0.00044 per K (Ford, *et al.*, 2003).

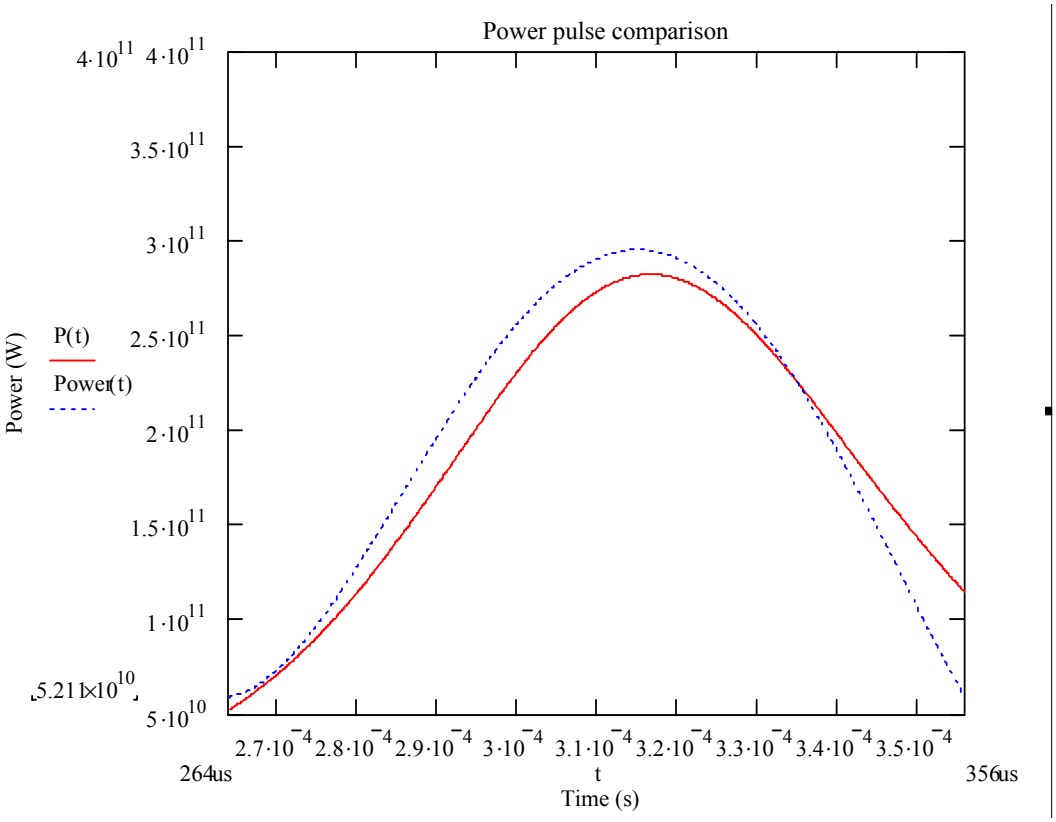


Figure 6.10: Power profile comparison for $1.136 pulse

Full-width-half-maximum for the theoretical pulse illustrated here is ~65.44 μs, while the FWHM for the iteratively produced pulse is ~62.19 μs.

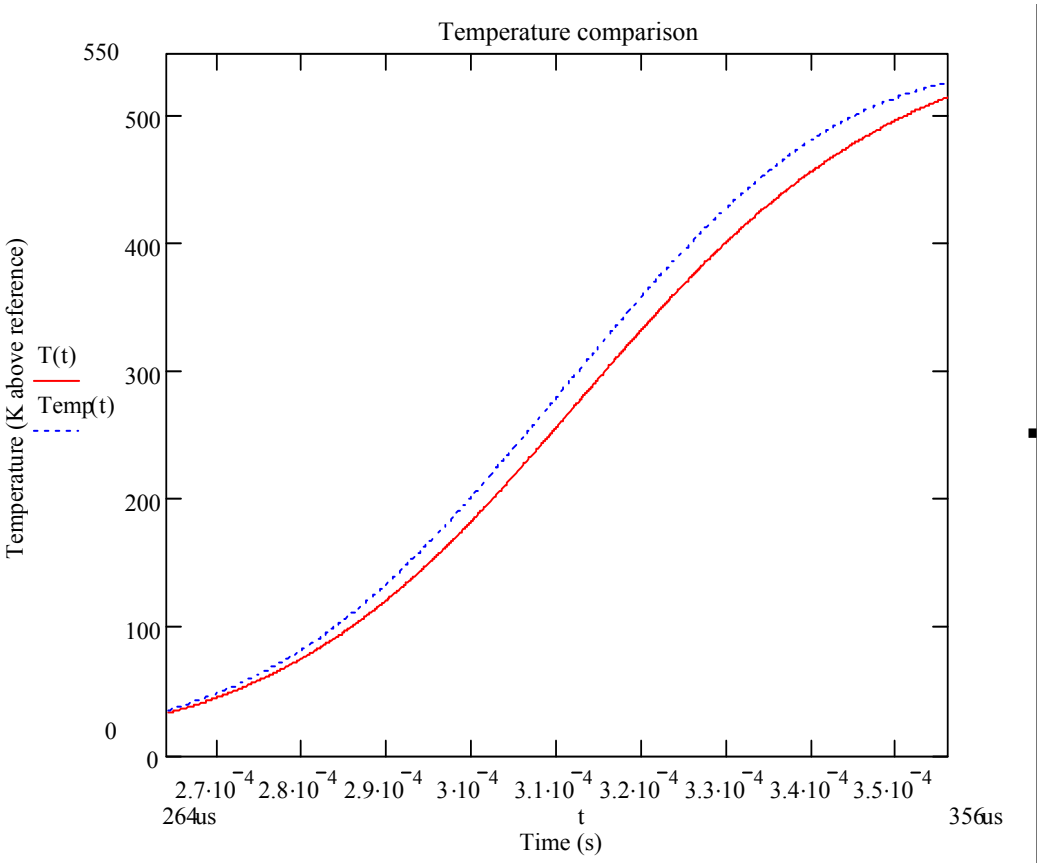


Figure 6.11: Temperature profile comparison for $1.136 pulse

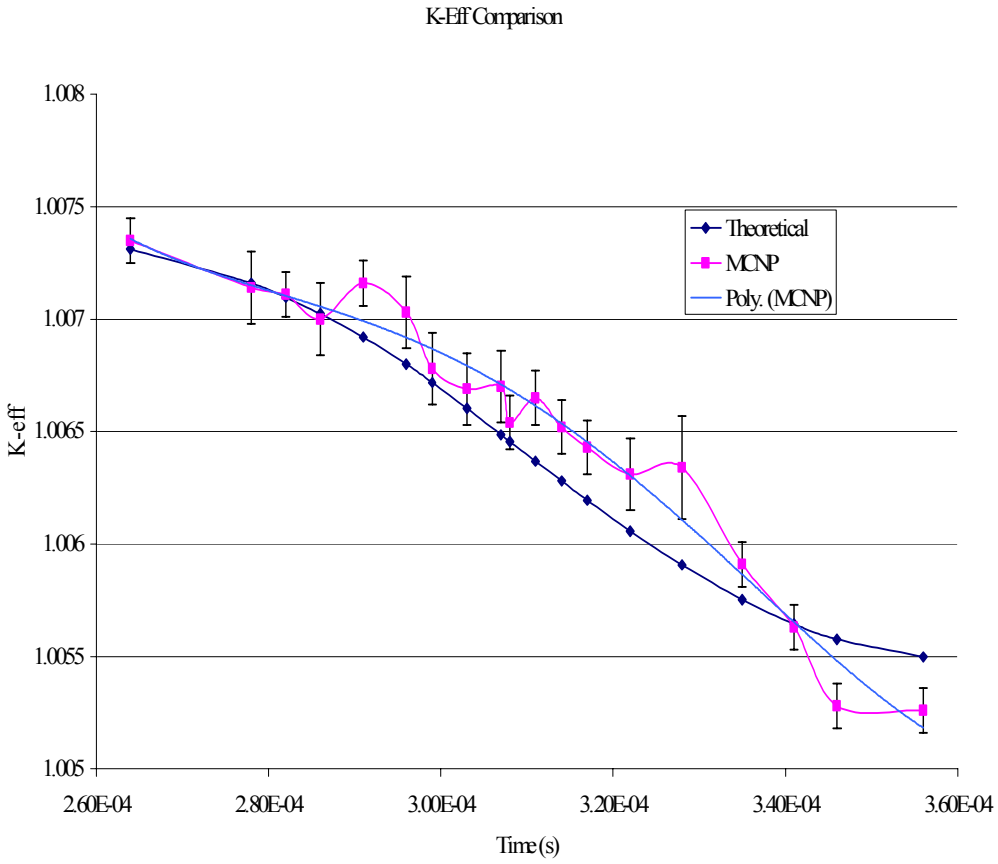


Figure 6.12: K-eff comparison for $1.136 pulse

Table 6.4 summarizes these kinetics results.

Time (s)	Power (W)	delta T (K)	MCNP k_{eff}	MCNP stdev
0.00E+00	0.00000E+00	0.00	1.00744	0.00014
2.64E-04	5.82093E+10	34.85	1.00735	0.0001
2.78E-04	1.21283E+11	75.67	1.00714	0.00016
2.82E-04	1.41925E+11	92.98	1.00711	0.0001
2.86E-04	1.64775E+11	112.78	1.007	0.00016
2.91E-04	1.91526E+11	140.86	1.00716	0.0001
2.96E-04	2.34641E+11	173.45	1.00703	0.00016
2.99E-04	2.58331E+11	195.56	1.00678	0.00016
3.03E-04	2.74990E+11	226.71	1.00669	0.00016
3.07E-04	2.85872E+11	258.58	1.0067	0.00016
3.08E-04	2.88849E+11	266.60	1.00654	0.00012
3.11E-04	2.88705E+11	290.47	1.00665	0.00012
3.14E-04	2.94895E+11	314.12	1.00652	0.00012
3.17E-04	2.93587E+11	337.51	1.00643	0.00012
3.22E-04	2.82915E+11	374.77	1.00631	0.00016
3.28E-04	2.58093E+11	415.46	1.00634	0.00023
3.35E-04	2.35101E+11	457.41	1.00591	0.0001
3.41E-04	1.83117E+11	486.98	1.00563	0.0001
3.46E-04	1.35588E+11	505.39	1.00528	0.0001
3.56E-04	5.90153E+10	526.52	1.00526	0.0001

Table 6.4: Summary of numerical kinetics results for $1.136 pulse

6.3 OTHER RESULTS

We present a plot of the temperature distribution for the SPR III core as obtained from fission energy deposition tallies in MCNP. Since temperature rise is a direct result of fission energy deposition in this system, it may be assumed the fission energy deposition and temperature distributions are the same. It was found that this distribution did not change significantly over the course of either pulse. Given the magnitude of the calculated displacements compared to reactor dimensions, the lack of variation in this distribution over time is understandable.

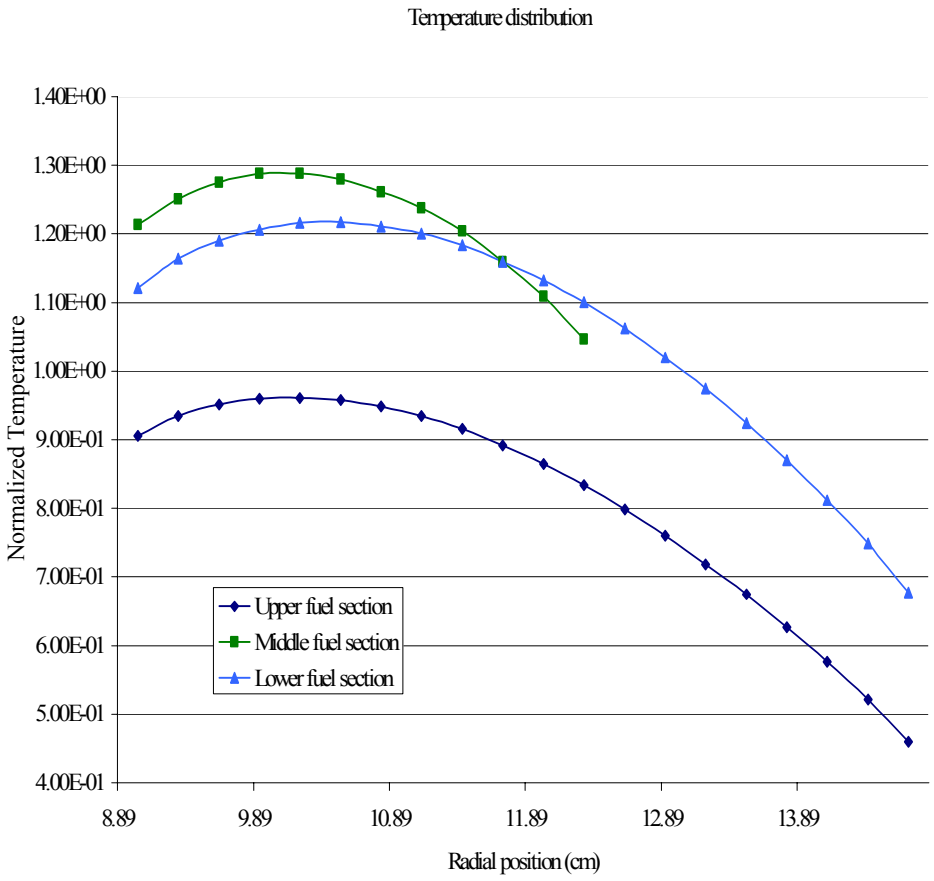


Figure 6.13: Temperature distribution during SPR-III pulse

The temperature derivative term in the thermoelastic radial displacement equation requires a polynomial fit to this spatial distribution multiplied by the magnitude in time obtained via Nordheim-Fuchs. While allowing this term to go to zero still allows for an oscillation due to the temperature terms in the zero-stress boundary conditions at the inner and outer surfaces, information is lost by doing so. The temperature is scaled in this plot so the spatially averaged temperature rise is equal to unity. The upper fuel section is always cooler that the lower due to system specific reflection properties in the SPR III geometry.

Finally, we present a summary of the average shutdown coefficient for both pulses, compared to that used for the associated theoretical fit.

	Theoretical Fuel Temperature Coeff ($/K)	*Average Iterative Fuel Temperature Coeff ($/K)*	*Population Stdev ($/K)*
$1.0846 Pulse	-0.0007040	-0.0007626	0.0002807
$1.136 Pulse	-0.0004400	-0.0004797	0.0000944

Table 6.5: Summary of shutdown characteristics for different pulses

Note that the standard deviation presented for the iterative results is only a population standard deviation. No uncertainty is carried forward from MCNP. This is explained shortly. It is worth noting the close agreement between the average values obtained iteratively and the theoretical values fitted to observed behavior.

6.4 SPHERICAL SIMULATION

Power, temperature and displacement results for a hollow sphere with a small internal cavity were also generated using the MOL/MCNP method. These results may be compared to those generated for a solid sphere by the code *ODMain* seen in Section 7.3. They are presented here without comparison to a Nordheim-Fuchs fit, as operational data for a spherical assembly consisting of U-10 Mo is not available. Figure 6.15 shows the power behavior as a function of time for a reactivity addition of approximately $1.15. This addition is estimated using SPR II reactivity data.

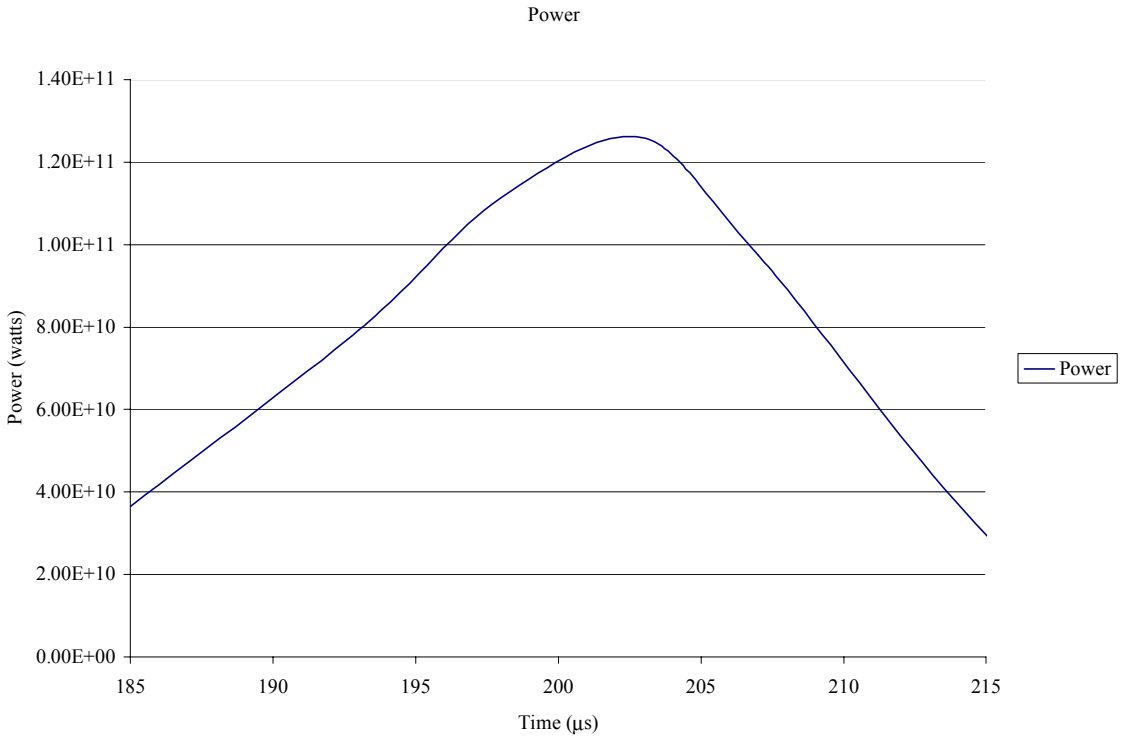


Figure 6.14: Power curve for $1.15 pulse

While retaining a fairly typical profile for prompt excursions, the power curve in Figure 6.14 has some slight hitches. This is due to the relatively low number of iterations run for this simulation compared to the much smoother finite element results presented later. Figure 6.15 shows the temperature behavior for this simulated excursion.

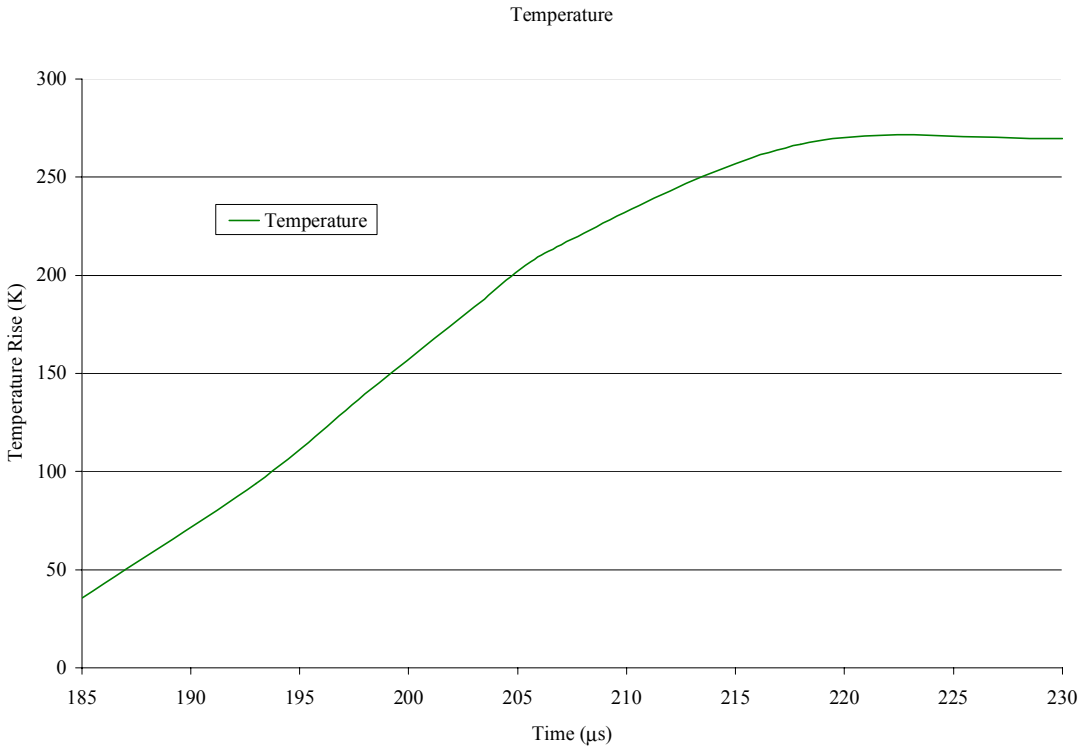


Figure 6.15: Temperature curve for a $1.15 pulse

Figure 6.15 does not exhibit any unusual behavior other than that expected from the slightly atypical power rise shown in Figure 6.14. Again, this is due to the relatively low number of iterations performed for this simulation. Finally, Figure 6.16 shows the highly unusual expansion behavior expected during excursions of this kind.

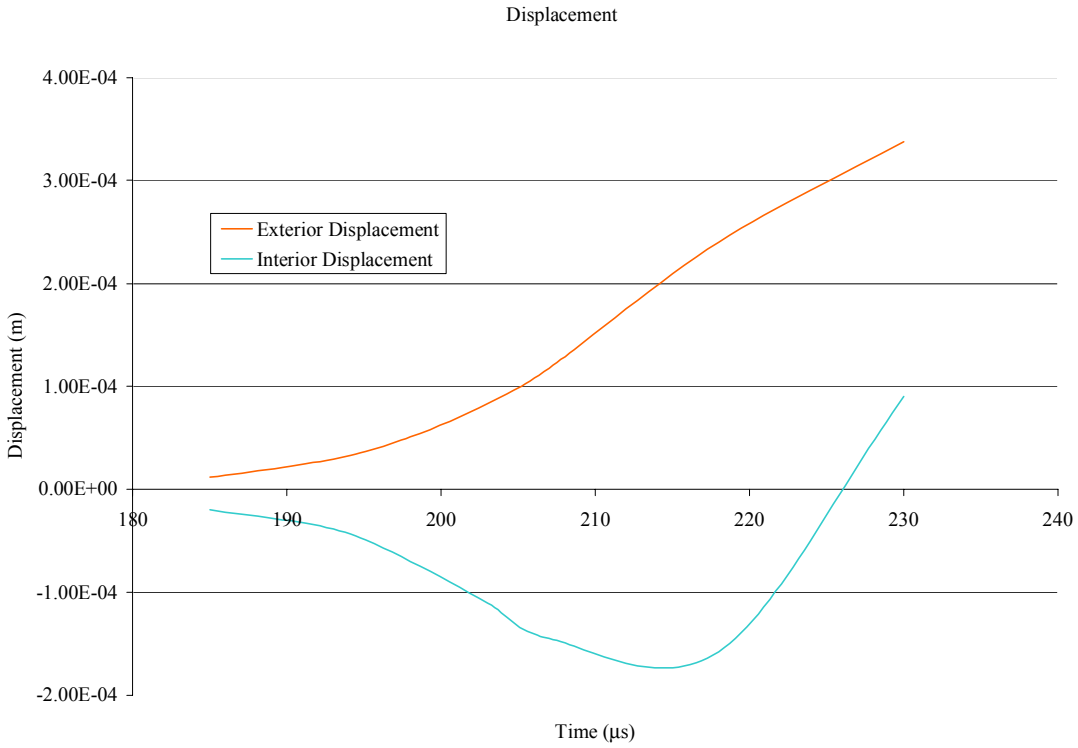


Figure 6.16: Displacement curve for a $1.15 pulse

We see in Figure 6.16 the displacement at the interior surface shocks inward strongly, then returns to positive displacement, where it will go on to oscillate about its equilibrium position. Both displacement curves exhibit the significant inertial delay inherent in the behavior of these reactors.

6.5 Discussion of Initial Results

The agreement between the SPR III observed data fit to the Nordheim-Fuchs kinetics model and the method-of-lines/MCNP power, temperature, and k_{eff} curves for two significantly different pulse sizes demonstrates the effectiveness of the method employed.

First, we shall discuss the thermoelastic radial displacement behavior of both pulses. One weakness in the analysis is that no displacement data for SPR III were ever collected. However, extensive experimental work was performed on SPR II (Reuscher, 1969, 1972). The similarity in geometry and the identical fuel material allow us to make comparisons.

The most striking feature of the radial displacement during a pulse is that the interior surface tends to move *inward*. A ring subjected to a thermal transient typically moves outward. Observing Figures 6.1, 6.2, 6.3, and 6.7, 6.8, and 6.9, one can see that only the middle section of fuel, with its smaller outer radial dimension, under the relatively slow thermal transient of a small pulse, behaved as an annulus of isotropic, homogeneous material does under more normal circumstances of thermal expansion.

In fact, this unusual behavior is due to and helps illustrate the extreme thermal transients affecting the fuel. So much thermal energy is deposited over such a small time scale that the radial stresses produced are large enough to briefly overcome the angular (hoop) stresses induced by expanding both inward and outward. This behavior does not continue beyond the initial energy deposition, but contributes significantly to the overall criticality change in the system and cannot be neglected in a criticality calculation.

This behavior is exaggerated during the faster, larger pulse. This agrees with the idea that only extremes in both energy deposition and time scale can produce these kinds of results.

Reuscher's experimental work on SPR II, in which he measured the interior and exterior radial displacements in time via transducer, agree with these theoretical results for SPR III (Reuscher, 1969, 1972).

The numerical neutron kinetics results agree strongly with the symmetric Nordheim-Fuchs kinetics model. In fact, they represent a slight improvement: it has been

well documented that fast burst reactor pulses toward the higher edge of the operational range of reactivity insertions tend toward the asymmetric power profile observed in the simulated $1.136 pulse presented here (Hansen, 1952); (Burgreen, 1962).

Observing the thermoelastic displacement and power in time simultaneously explains the asymmetry in the power profile. The criticality effects of the inertial delay in fuel expansion are felt most strongly at the beginning of the pulse, when the fuel has barely begun to move. Towards the end of the pulse, the fuel is moving quickly towards its equilibrium displacement for non-dynamic thermal expansion, i.e., where it will come to rest seconds after the pulse termination. Observing the k_{eff} curves in Figures 6.6 and 6.12 shows a delay in reactivity drop that corresponds directly to the delay in fuel expansion illustrated in every displacement curve.

This expansion behavior also explains why the asymmetry of the power profile is far more noticeable in the large, $1.136 pulse than in the smaller $1.0846 pulse. During the faster pulse, we see the power rise to a point higher than that predicted by theory, then drop off more quickly, as illustrated by the smaller numerical FWHM value. This is also the case in the slower pulse, but the effect is hardly noticeable, and the power profile much closer to symmetric. Both behaviors derive directly from the displacement lag in time. As the initial reactivity insertion increases, the inertial delay becomes relatively more significant, and the effective fuel temperature coefficient of reactivity drops in magnitude.

Therefore, the iterative method employed here not only closely agrees with a theoretical fit of observed SPR III data, but also represents an improvement over that fit in terms of describing SPR III behavior during a pulse.

6.6 UNCERTAINTY AND ERRORS IN NUMERICAL RESULTS

Tables 6.2 and 6.4 both present standard deviations resulting from statistical MCNP calculations. The results presented thus far, however, are without appropriate uncertainty propagation.

By changing the initial step reactivity insertion slightly (on the order of 0.00001), one modifies the resulting power profile, in time if not in magnitude and shape. Therefore, we assume that the first k_{eff} obtained from MCNP is correct, and that we may neglect its associated standard deviation.

While this assumption would be entirely invalid if the only source of reactivity information were MCNP, it can be made using experimental information on differential burst element reactivity worth (Ford *et al.* 2003). Using this information, we position the burst element in the MCNP geometry to insert exactly the desired amount of reactivity.

A rigorous treatment of uncertainty propagation in the situation of a self-adaptive numerical system is difficult. A qualitative explanation is as follows: if a k_{eff} higher than the correct value is produced, the next iteration is heated to a higher temperature than is correct, producing larger displacements and smaller densities, leading to a *smaller* k_{eff} value for the next iteration. Therefore, the k_{eff} curve tends to "dance" around what is physically correct for that reactivity addition. Thus the system can be described as self-correcting to a degree. By observing figures 6.6 and 6.12, we see this is precisely the case. The way to produce a truly incorrect curve is to skimp on the number of time iterations, especially near the power maximum. This increases the time a "bad" k_{eff} value has to wildly change the power profile. Characterizing this tendency exactly for the purpose of uncertainty propagation is beyond the scope of this work.

Finally, we are in the fortunate position of having experimental data available for SPR III. While displacement data were never taken, extensive in-core temperature data

acquired via thermocouples are available, and were used in the construction of Table 2.2. This table gives us the experimental temperature rise, and Nordheim-Fuchs gives a temperature-appropriate power profile. Since our iterative calculations were independent of a fuel temperature coefficient of reactivity, we can compare the differences between theoretical-fit and iterative temperature curves for each pulse to illustrate how closely they agree and where they differ.

Chapter 7: Finite Element Results and Discussion

7.1 SPR II SIMULATION RESULTS

In this chapter we present results produced by the finite element codes developed during this project. They run approximately twenty times faster than the method-of-lines/MCNP numerical method described in Chapter 4. The transport solution via diffusion approximation is less correct, but the thermoelasticity solution is fully two-dimensional instead of a solution of an approximate one-dimensional equation.

Operational data for SPR II extends to its displacement behavior as a function of time. For the purposes of simulation, this provides a good opportunity for benchmarking. SPR II neutronics, temperature and displacement results at various points on the finite element mesh shall be presented and compared to known behaviors. Two pulses, one fast and one slow, will be presented in order to demonstrate the significance of inertial lag in reactor behavior and the successful reproduction of the effect by the code.

7.1.1 SPR II Power and Temperature Results (Small Pulse)

The neutronics behavior of the SPR II reactor is intimately tied to the thermomechanical feedback of the fuel mass. This behavior consists of an inertial lag in fuel expansion as well as highly unusual expansion characteristics observed mostly in more intense pulses. First we present the power and temperature results for a relatively low power pulse. Figure 7.1 shows the power as a function of time for a computationally produced pulse as well as a Nordheim-Fuchs point kinetics fit of operational SPR II data:

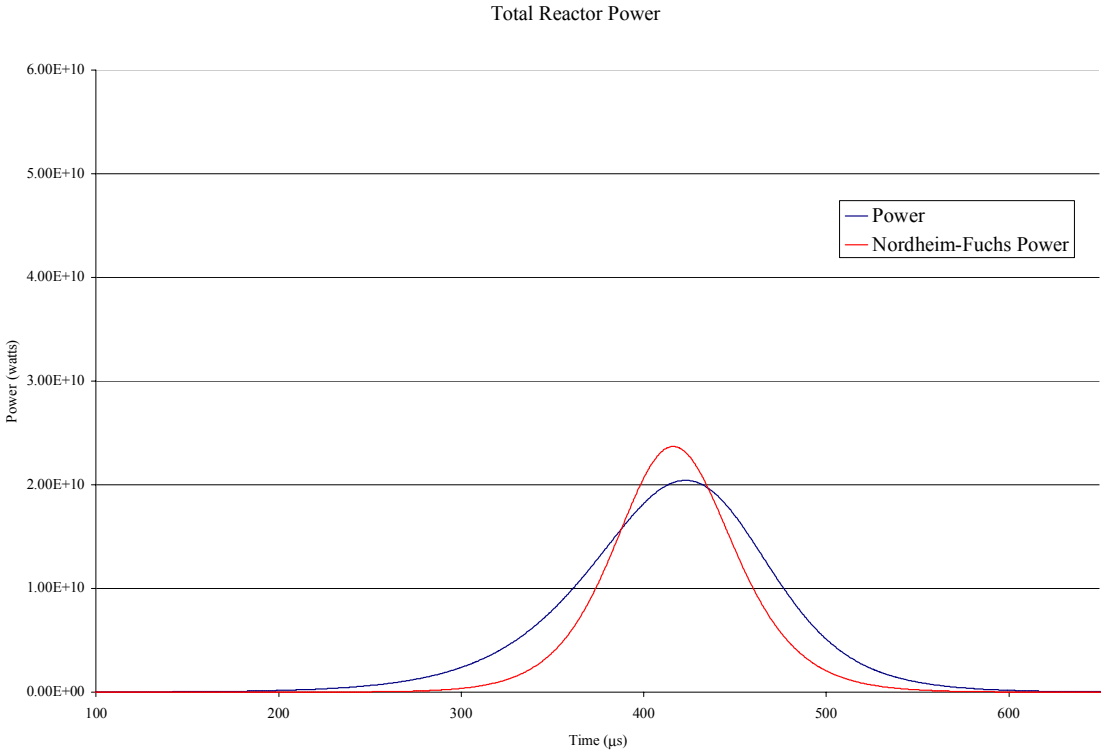


Figure 7.1: Power comparison for a small (~7.55 cents prompt) addition

Figure 7.1 shows the Nordheim-Fuchs fit to be a sharper curve, whereas the computational results give a lower, broader profile. The parameters for the Nordheim-Fuchs fit for this small reactivity addition are as follows: the initial power is 200 W and the temperature coefficient of reactivity is -$0.0006 per K. These data are consistent with previous models of SPR behavior (Ford *et al.*, 2003).

Figure 7.2 gives the same comparison for temperature rise as a function of time:

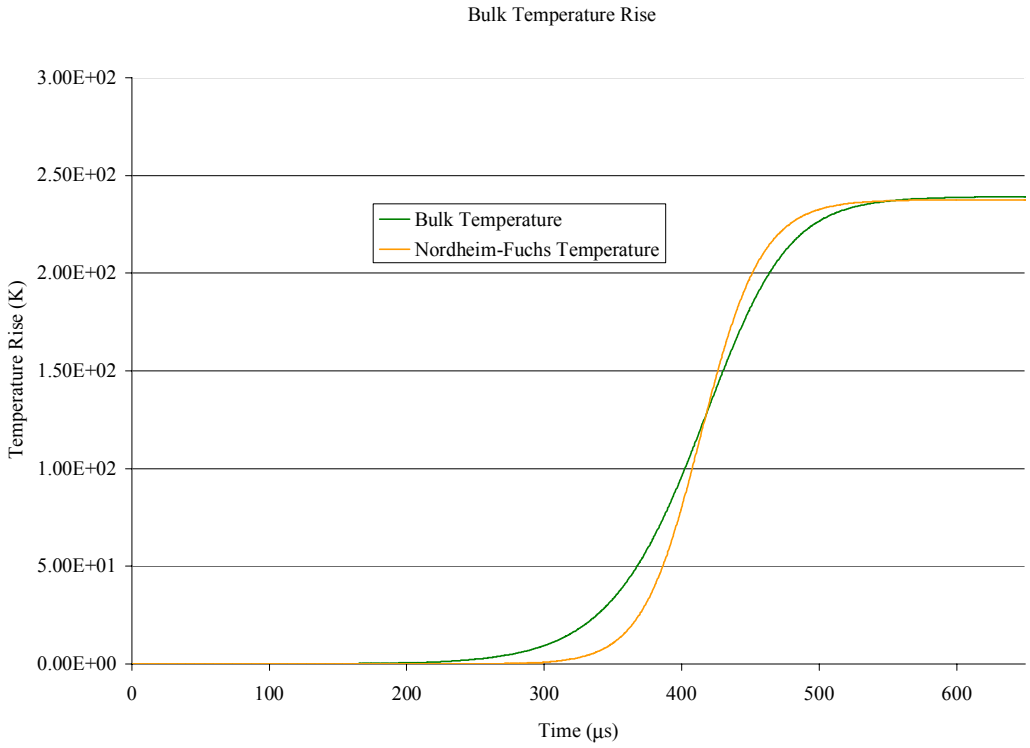


Figure 7.2: Temperature comparison for a small (~7.55 cents prompt) addition

Figure 7.2 shows the bulk temperature rise for the entire reactor mass. The Nordheim-Fuchs theoretical fit of operational data provides only this bulk temperature rise. The temperature produced computationally provides a temperature distribution, but is mass-averaged in Figure 5.2 for the purpose of comparison.

The results shown in Figures 7.1 and 7.2 are relatively poor. The reason for this is dynamic, but may be attributed mostly to the one-group diffusion approximation of the reactor behavior. This approximation was chosen to keep an already complicated code as simple as permissible. While it provides decent results for this small reactivity addition, they are not outstanding. This is primarily due to the need to select a single neutron velocity averaged across the entire fission spectrum. This neutron velocity directly

controls how fast the system power ramps up in time (the inverse of the neutron velocity is the sole integrand in the diffusion mass matrix). A higher value for neutron velocity produces a sharper power profile, and vice versa. However, changing the neutron velocity from pulse to pulse while simulating the same reactor system is physically inappropriate; this option is not utilized.

7.1.2 SPR II Displacement Results (Small Pulse)

The displacement of the SPR II fuel mass during a small pulse is more typical of common hoop element expansion than that observed during a large pulse. Figure 7.3 shows displacements at the four corners of the solid region mesh:

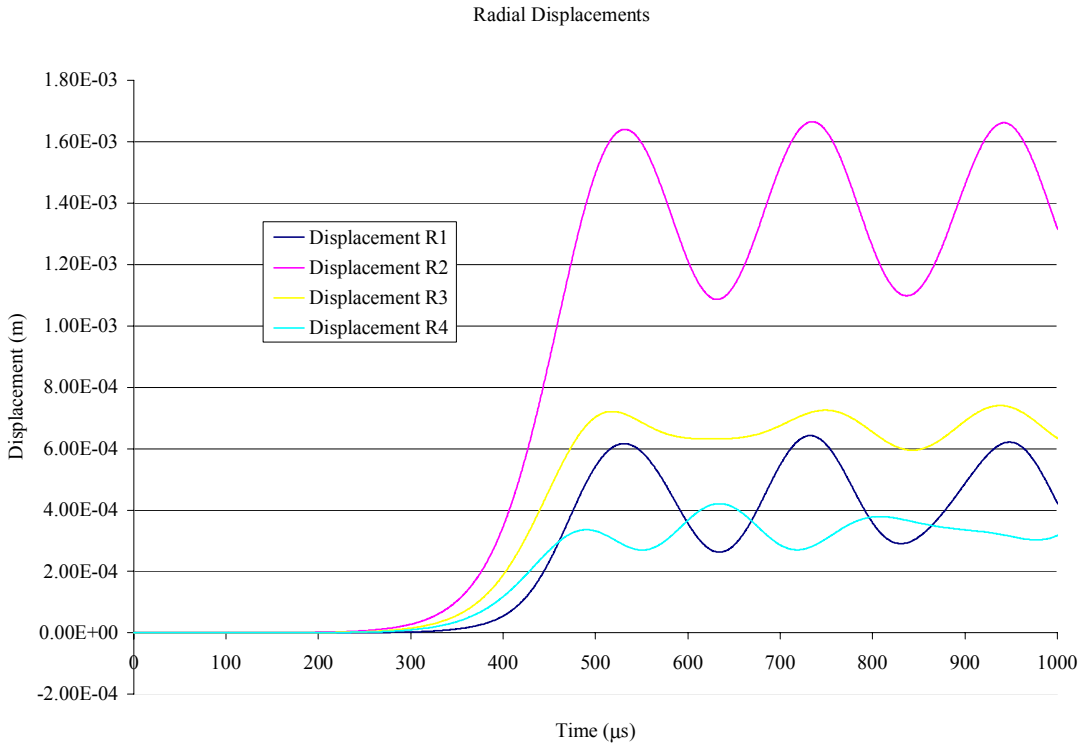


Figure 7.3: Radial displacement during a small (~7.55 cents prompt) addition

The four points at which displacement is displayed in Figure 7.3 are as follows: R1 gives displacement at the lower left corner of the solid region, R2 at the lower right, R3 at the upper left, and R4 at the upper right. The results are fairly typical of an expanding hoop element: initial expansion to a point slightly past equilibrium followed by oscillation about the equilibrium position. The differences in expansion behavior seen in Figure 7.3 may be attributed to the spatially dependent temperature field and the magnitude of the radial position at which each displacement is tracked.

Figure 7.4 shows the accompanying axial displacement tracked at two points on the upper surface of the cylinder.

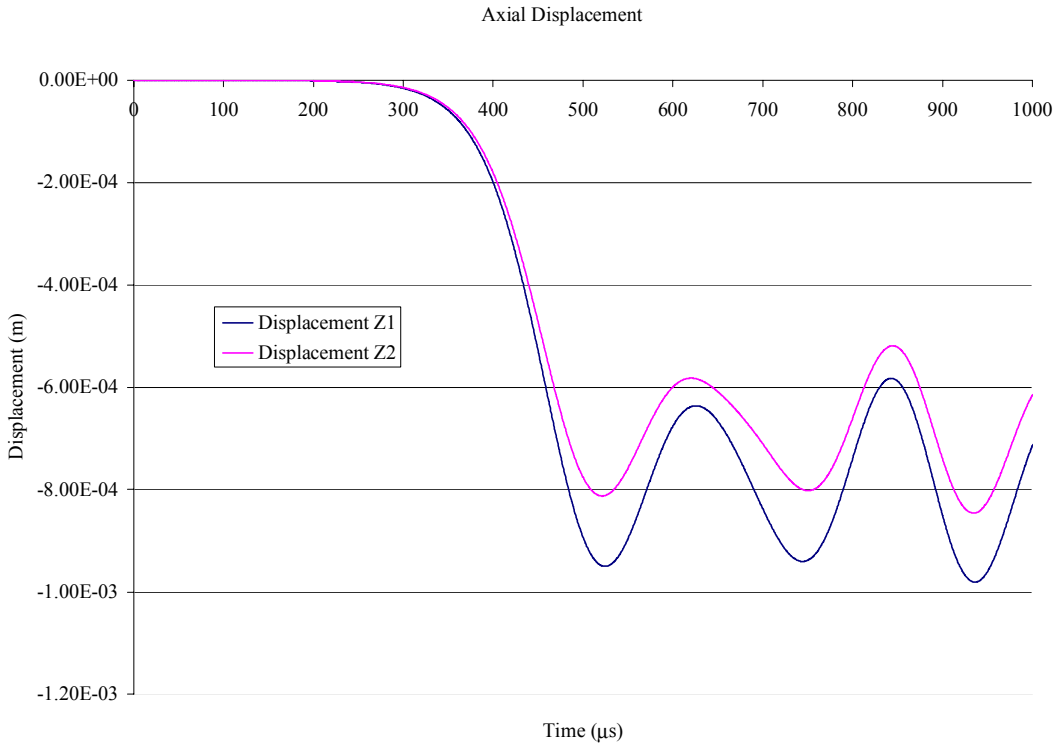


Figure 7.4: Axial displacement during a small (~7.55 cents prompt) addition

The two points where displacement is tracked in time in Figure 5.4 are Z1 at the upper left corner of the solid mesh, and Z2 at the upper right. The reason for the negative axial expansion lies in a major assumption required to approximate SPR behavior.

The thermoelasticity behavior modeled by the solver built into the *TDMain* code is that of a single solid piece of material, instead of many plates stacked axially. Complicating the solid model to incorporate the interface conditions between fuel plates is beyond the scope of the project. However, an assumption of zero axial traction as shown by Wimmet in his analytical solution of the one-dimensional approximation of SPR displacement behavior provides accurate feedback for the neutronics behavior (Wimmet, 1992).

The displacements observed in Figure 5.4 may be interpreted as the net axial expansion seen in a number of plates stacked one atop the other during a period of significant radial expansion.

7.1.3 SPR II Spatial Flux Profile (Small Pulse)

Figure 7.5 shows the radial flux profile at the centerline of the reactor and at the top. It is worth noting that the change from the beginning of the pulse to the end of the pulse is insignificant.

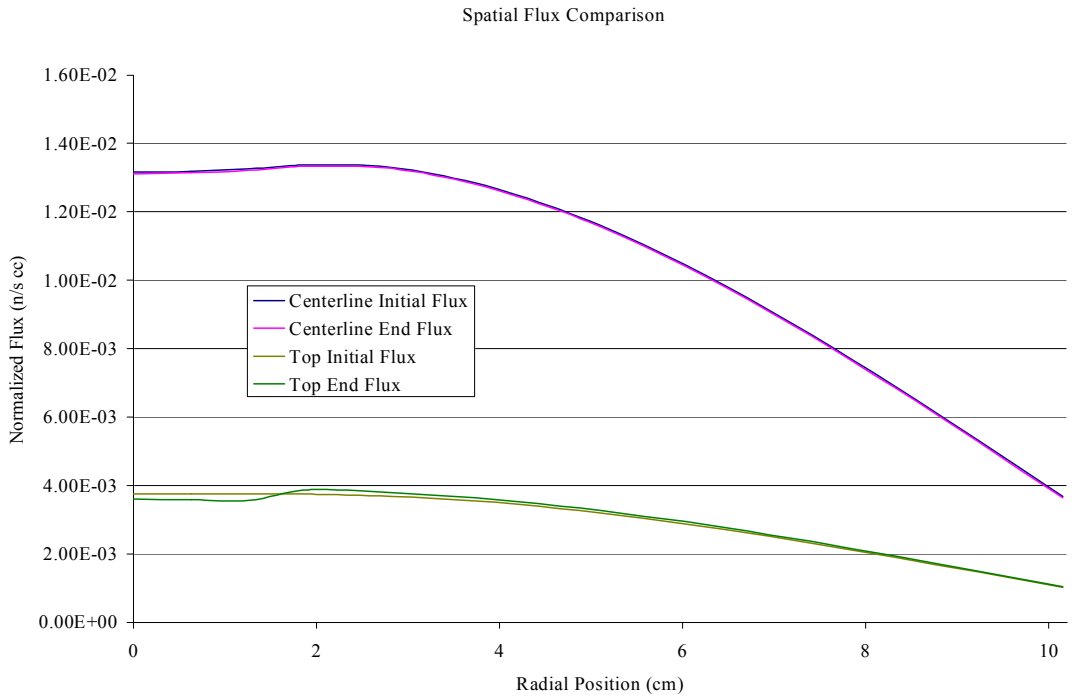


Figure 7.5: Spatial flux profile during a small (~7.55 cents prompt) addition

In terms of normalized flux, it is clear that the neutron flux along the radial centerline is much more intense than that along the top of the reactor. The changes in profile are virtually nonexistent in the case of a small pulse.

7.1.4 SPR II Power and Temperature Results (Large Pulse)

The time-dependent behavior of a larger SPR II pulse compares much more favorably to SPR II operational data. Figure 7.6 shows the same neutron power comparison seen in Figure 7.1, but for a larger prompt reactivity addition.

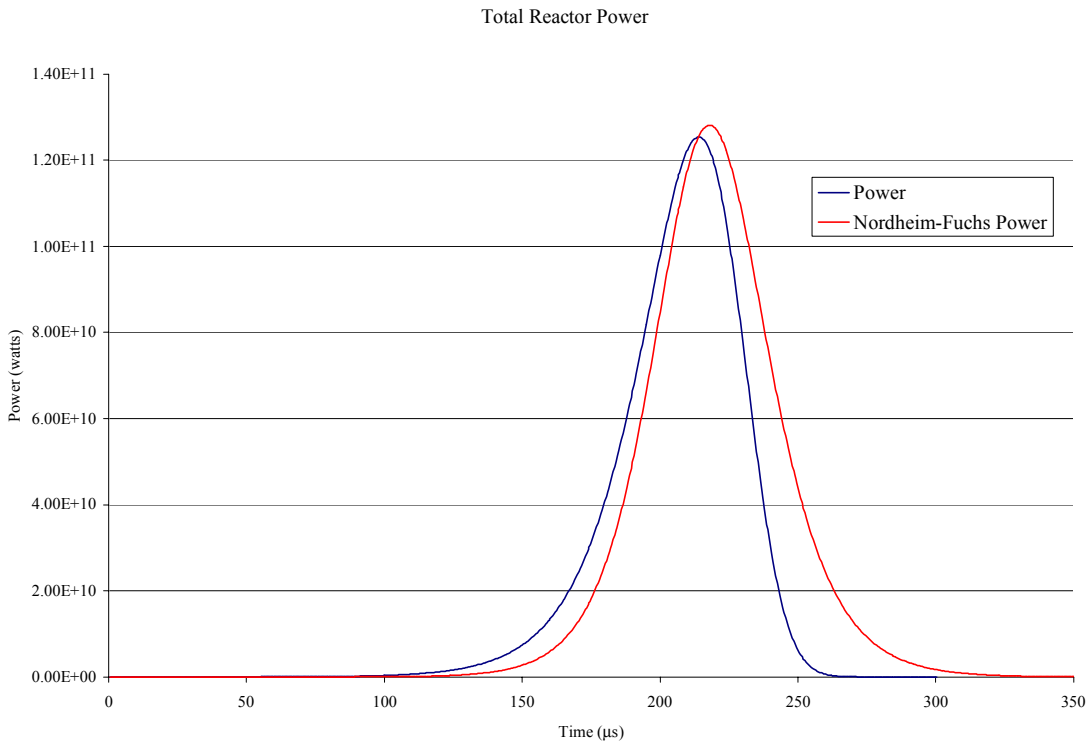


Figure 7.6: Power comparison for a large (~12.1 cents prompt) addition

Figure 7.6 strongly validates the numerical model employed. The reactor power magnitudes are very similar, and the computationally produced power curve has the asymmetric profile typical of larger pulses. This shows that the code is reproducing the inertial fuel lag correctly: the blue power curve in Figure 7.6 ramps up somewhat slowly,

and ramps down sharply at the end of the pulse. The fuel mass expansion has "caught up" by the end of the reactor pulse to shut down the excursion. The symmetric Nordheim-Fuchs fit to operational data is not capable of reproducing this behavior.

The parameters for the fit were 20 kW initial power and a temperature coefficient of reactivity of -$0.000466 per K. Note the difference between the temperature coefficients of reactivity used for the small and large pulses. They are consistent with the coefficients used by SPR II operators to distinguish between "non-inertial" and "inertial" behavior (Ford *et al*, 2003).

Figure 7.7 compares the temperature behavior in time for the computational data and a Nordheim-Fuchs fit of operational data:

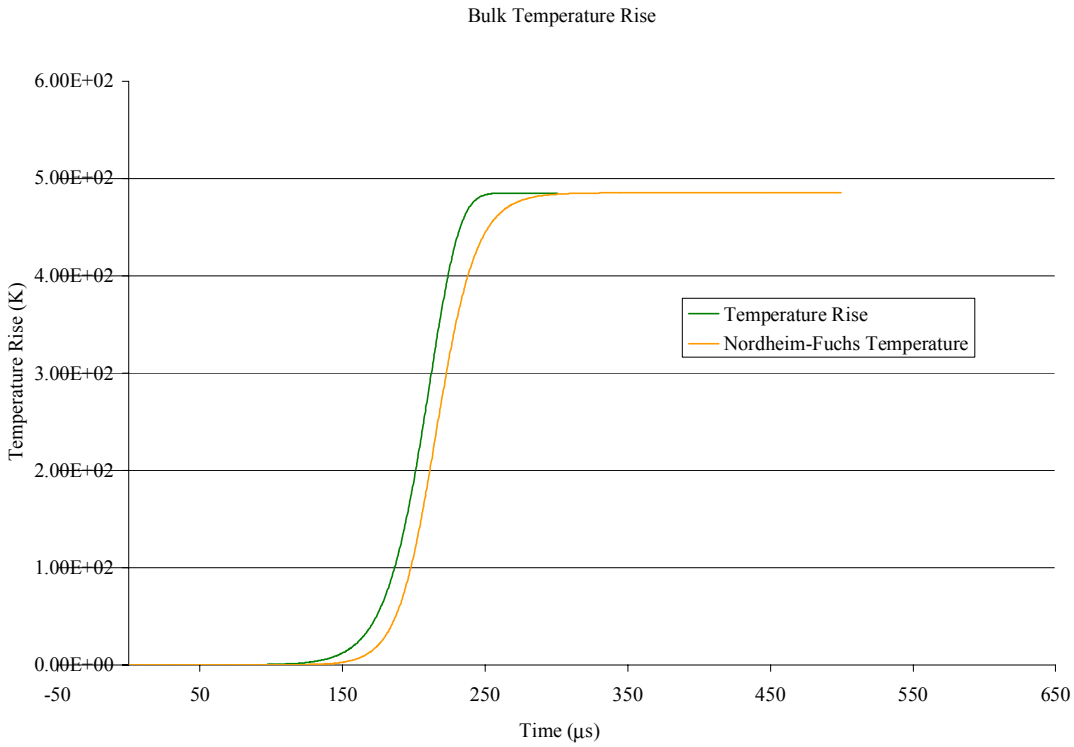


Figure 7.7: Temperature comparison for a large (~12.1 cents prompt) addition

Figure 7.7 shows the two temperature behaviors are very similar. The temperature data in time is observed during and immediately after reactor operation; the Nordheim-Fuchs power curve is a fit based on this data. The similarity between the operational temperature curve in Figure 7.7 and the accompanying computational curve is the strongest validator of the reproduced neutron behavior.

7.1.5 SPR II Displacement Results (Large Pulse)

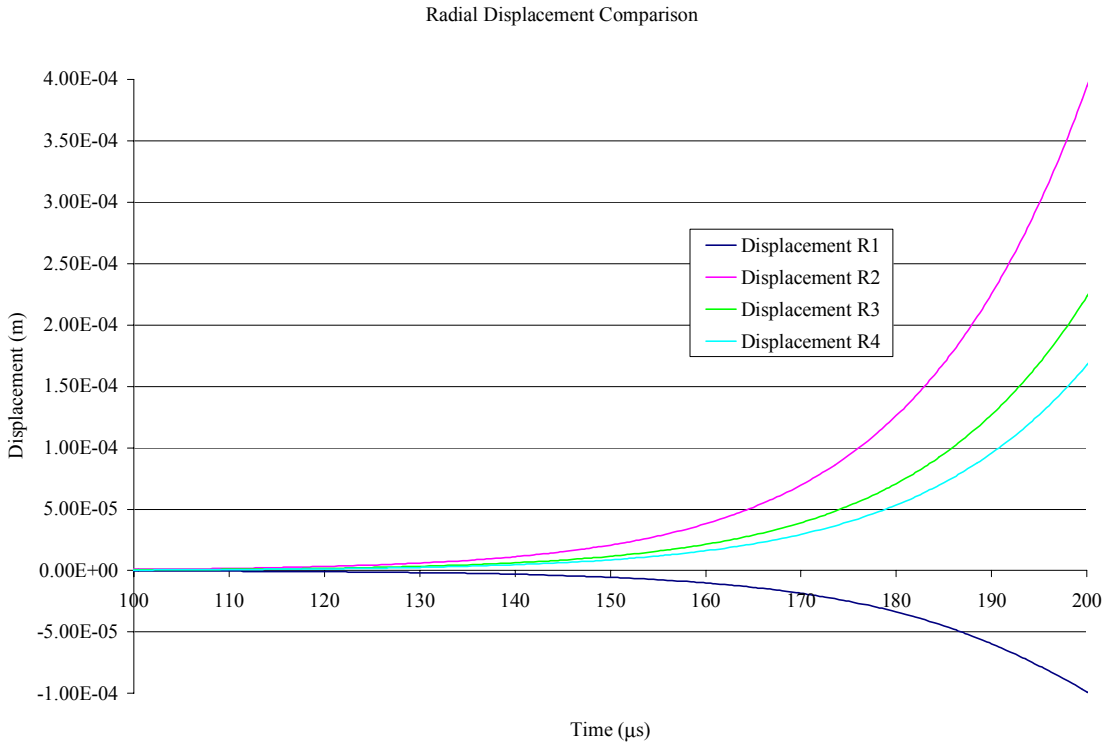


Figure 7.8: Radial displacement during a large (~12.1 cents prompt) addition

Figure 7.8 shows the displacement behavior of the same four points observed in Figure 7.3, except during the larger reactor pulse.

The radial displacement of the inner surface of the cylinder along the radial centerline (point R1) is essential in the reproduction of SPR behavior. During a "large" pulse, the interior surface of the reactor shocks inward instead of smoothly expanding outward. This behavior is atypical of hoop element expansion. The negative displacement observed in the R1 curve in Figure 7.8 shows the elasticity model built into code *TDMain* reproducing this behavior. Further, the maximum negative displacement of the R1 curve in Figure 5.8 falls directly in the 6 to 10 mils range observed via transducer in a series of SPR II pulses (Reuscher, 1974) (Wimmet, 1992).

7.1.6 SPR II Spatial Flux Profile (Large Pulse)

The radial flux profile during a "large" reactor pulse is worth observing in order to note the more significant change at the inner material interface. Figure 7.9 shows the centerline and top radial flux profiles at the beginning and end of a large reactor pulse.

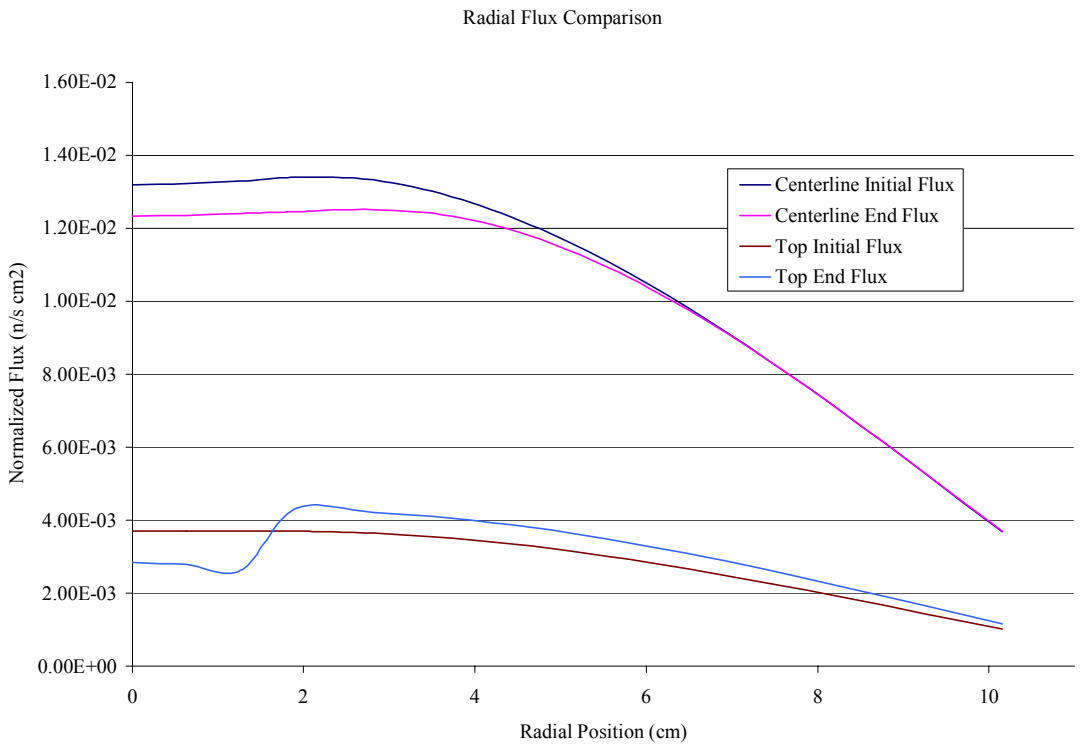


Figure 7.9: Spatial flux profile during a large (~12.1 cents prompt) addition

The "Top End Flux" shown in Figure 7.9 shows in particular the evolution of the flux profile as the solution progresses in time. The flat initial condition employed in the code moves to an end condition with a sharp flux drop at the interface between the solid interior surface and the air filling the cylindrical cavity. It is unusual to observe this accurate a reproduction of transport behavior in a diffusion solution near material boundaries. Doing so somewhat validates the neutron diffusion approximation for this reactor.

7.2 SPR III SIMULATION RESULTS

Here we present a comparable set of figures for the SPR III reactor. While displacement data were never taken during SPR III operation, behavior may be extrapolated from SPR II data. Power and temperature data will be compared to a Nordheim-Fuchs fit of operational data as in Section 7.1. Data for a mid-size and large pulse are presented to show the reproduction of the inertial fuel lag effect by the finite element solution.

Data produced during SPR III simulation is closer to the operational data than that produced for SPR II. This most likely derives from the simpler geometry of SPR III and the fact that it is more conducive to modeling via diffusion (more material relatively far from material surfaces).

7.2.1 SPR III Power and Temperature Results (Medium Pulse)

Figure 7.10 gives neutron fission power and temperature data for a pulse in the middle of SPR III's operational parameters.

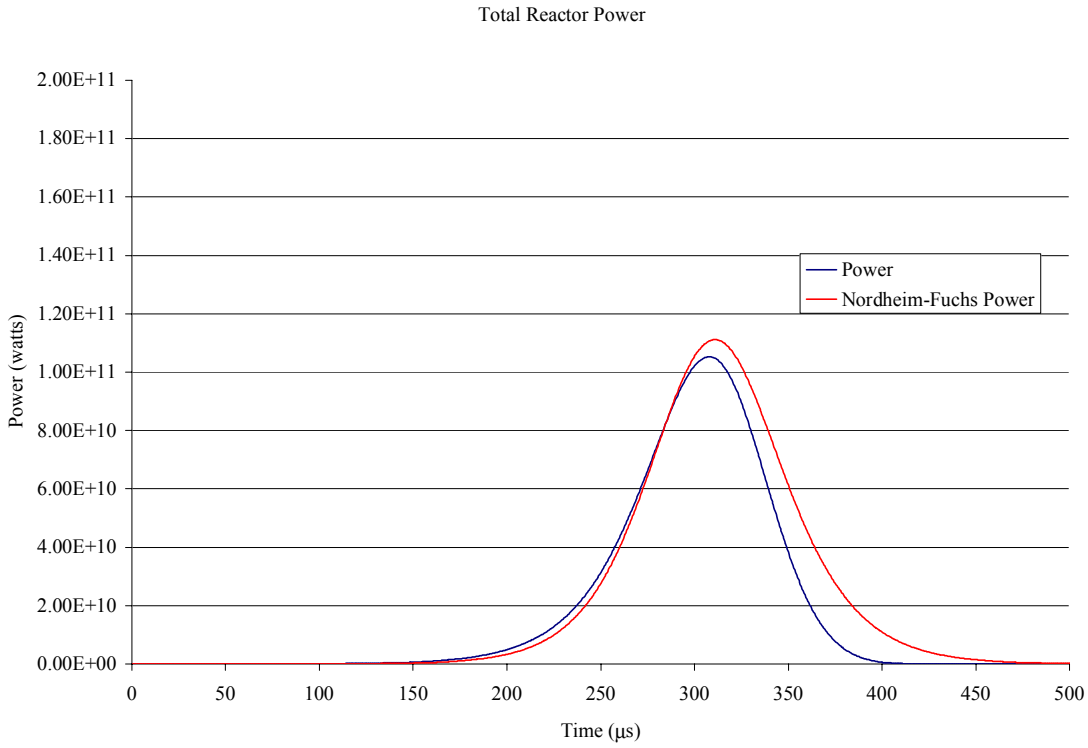


Figure 7.10: Power comparison for a medium (~10.3 cents prompt) addition

Figure 7.10 shows good agreement between operational and computational data. Some asymmetry is detectable in the blue curve; while it is not as obvious as in the case of a larger pulse, it still plays a significant role in the shutdown behavior of the reactor. The parameters for the red Nordheim-Fuchs fit are an initial power of 10 kW and a temperature coefficient of reactivity of -$0.000605 per K. Figure 7.11 gives the corresponding temperature behavior in time.

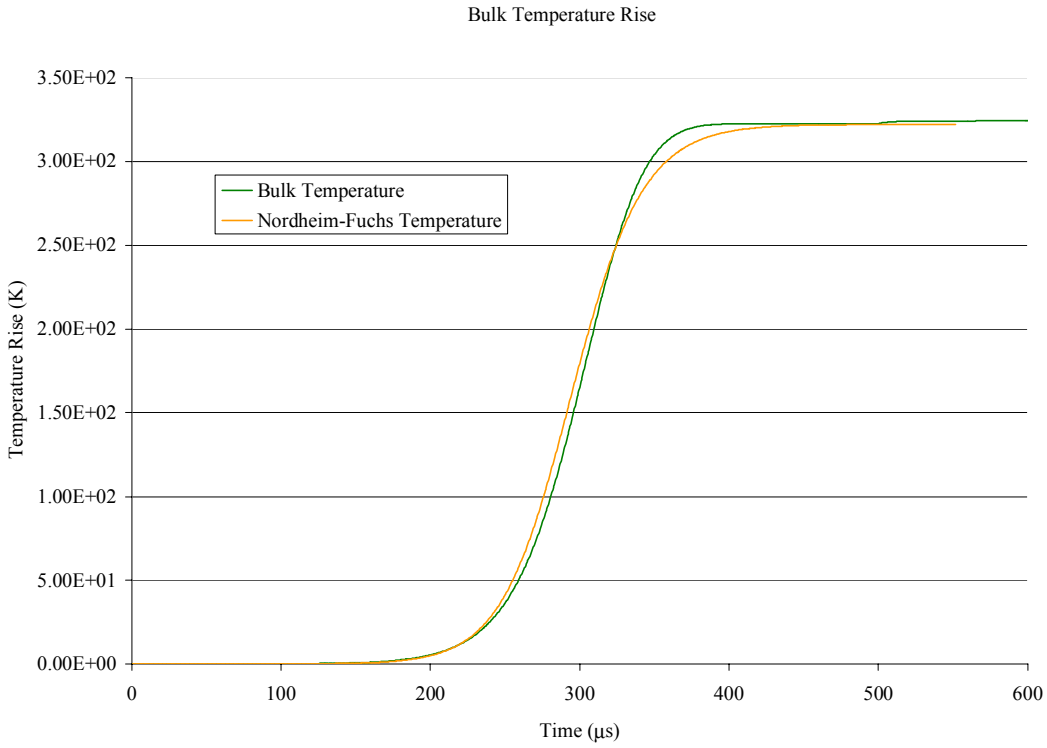


Figure 7.11: Temperature comparison for a medium (~10.3 cents prompt) addition

Figure 7.11 shows excellent agreement between computational and operational data. Accuracy to the degree shown above must be assumed serendipitous to some degree given the assumptions made during computational modeling, but nevertheless strongly validates the model.

7.2.2 SPR III Displacement Results (Medium Pulse)

Figure 7.12 gives the radial displacement as a function of time at the same four points on the solid mesh as presented previously for SPR II results.

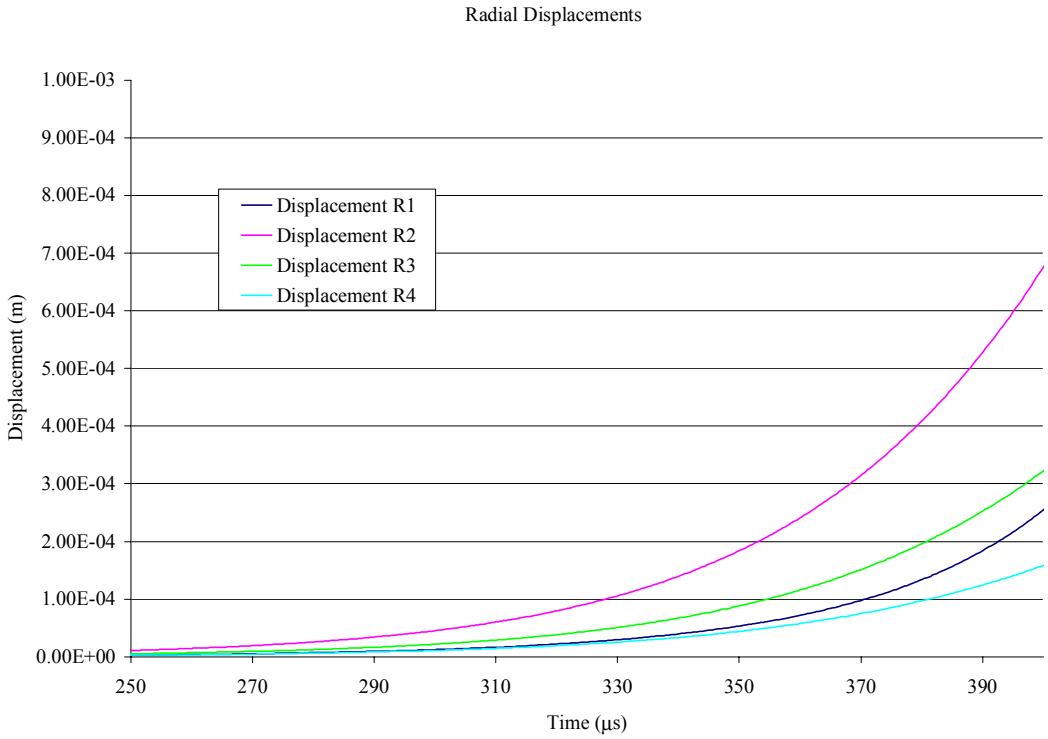


Figure 7.12: Radial displacement during a medium (~10.3 cents prompt) addition

While a significant shock inward is absent from the medium pulse simulation, an inertial lag in thermal expansion is certainly present. Observing Figure 7.11 and Figure 7.12 simultaneously shows the material's bulk temperature is already nearly at a maximum at 350 microseconds. At this point, the material displacement is at a fraction of its equilibrium position and is still in the initial expansion phase. Oscillation (not seen in Figure 7.12) is not reached until relatively long after the termination of the pulse.

7.2.3 SPR III Spatial Flux Profile (Medium Pulse)

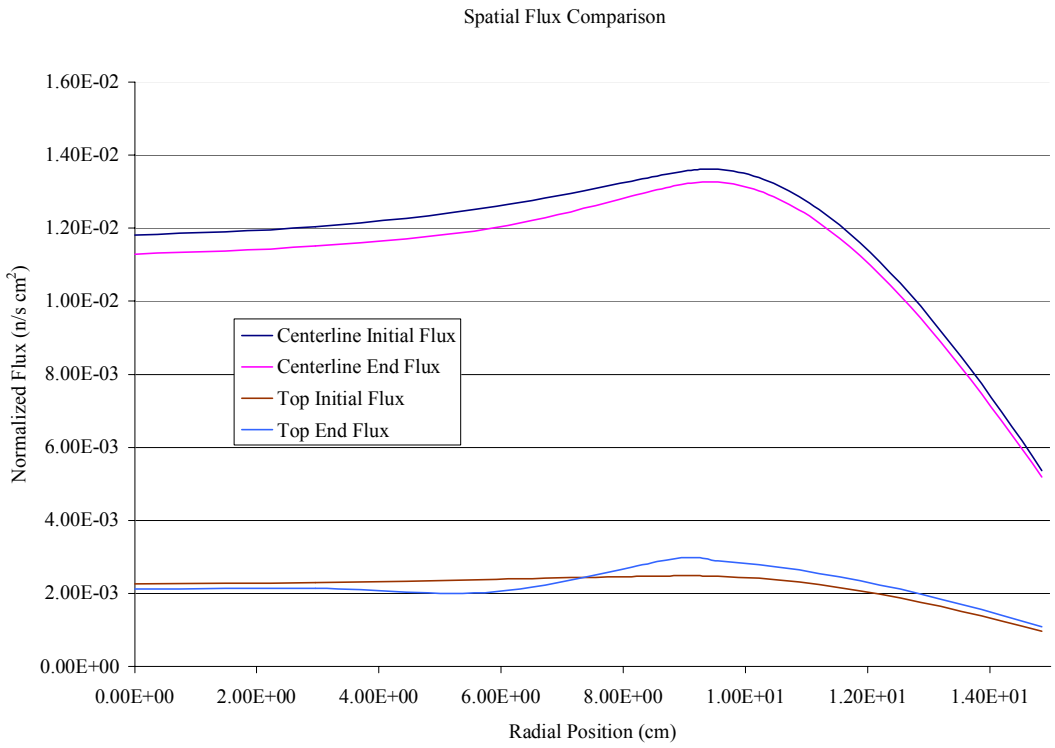


Figure 7.13: Spatial flux profile during a medium (~10.3 cents prompt) addition

Figure 7.13 shows again the evolution of the radial flux distribution. The numerical solution produces the desired flux shape across the material interface located at 8.89 cm. This is an excellent result for a neutron diffusion solution.

7.2.4 SPR III Power and Temperature Results (Large Pulse)

We present data comparisons for a pulse at the outer edge of the SPR III operational envelope in order to illustrate the clear reproduction of the inertial fuel behavior in our computational model.

Figure 7.14 shows numerical and operational data for a large SPR III pulse.

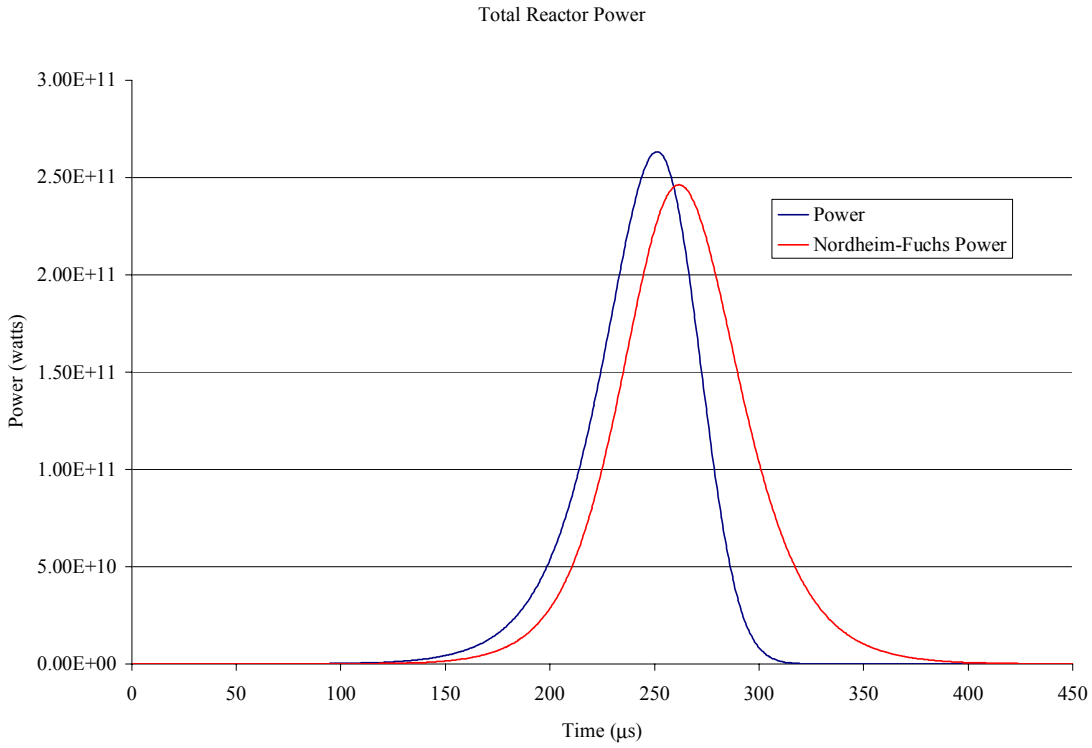


Figure 7.14: Power comparison for a large (~13.5 cents prompt) addition

Figure 7.14 once again illustrates the general agreement in power magnitude between the computational model and a symmetric fit of the operational data, as well as the disagreement in profile shape. The blue curve in Figure 7.14 clearly illustrates the asymmetry inherent in large pulse behavior for this reactor class. The parameters for the red Nordheim-Fuchs power curve were an initial power of 50 kW and a temperature coefficient of reactivity of -$0.000488 per K. The difference in temperature coefficient of reactivity between the large and medium SPR III pulses is consistent with values used by SPR III operators to predict reactor behavior based on the size of the reactivity addition (Ford *et al*, 2003). Figure 7.15 shows the corresponding temperature behavior.

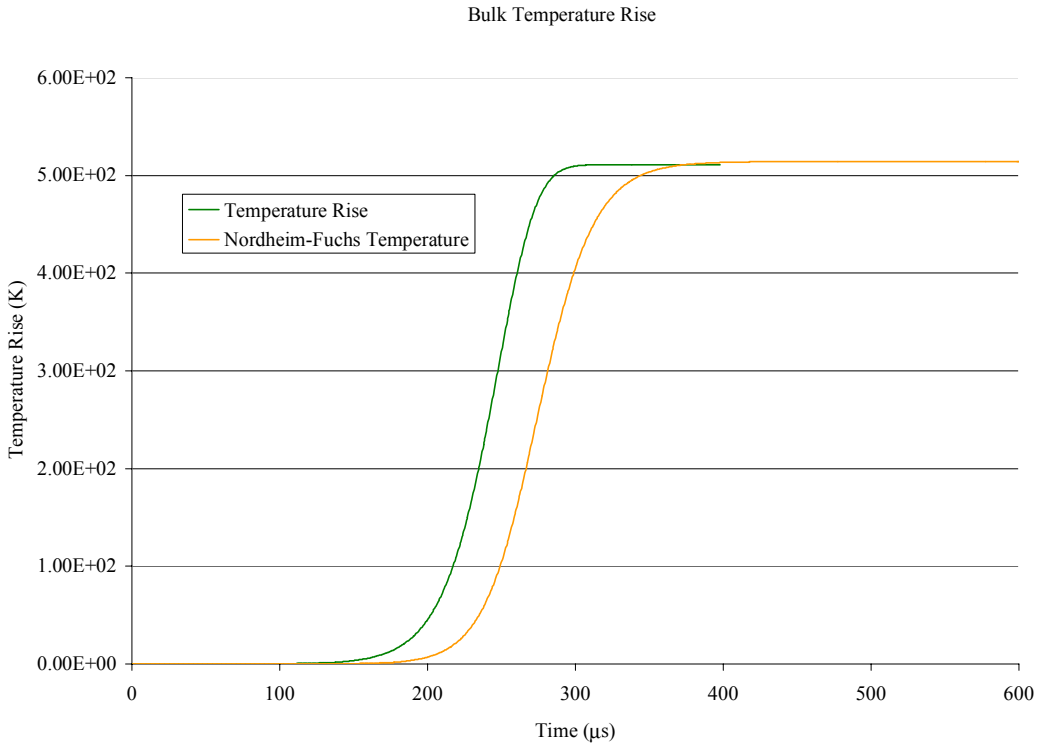


Figure 7.15: Temperature comparison for a large (~13.5 cents prompt) addition

The temperature comparison between computational and operational results differs significantly only in time position. The lack of profile differences means the simulation is substantially correct; a simple change of initial power condition would suffice to "move over" either curve in time.

7.2.5 SPR III Displacement Results (Large Pulse)

Figure 7.16 shows the radial displacement as a function of time for the large SPR III pulse modeled.

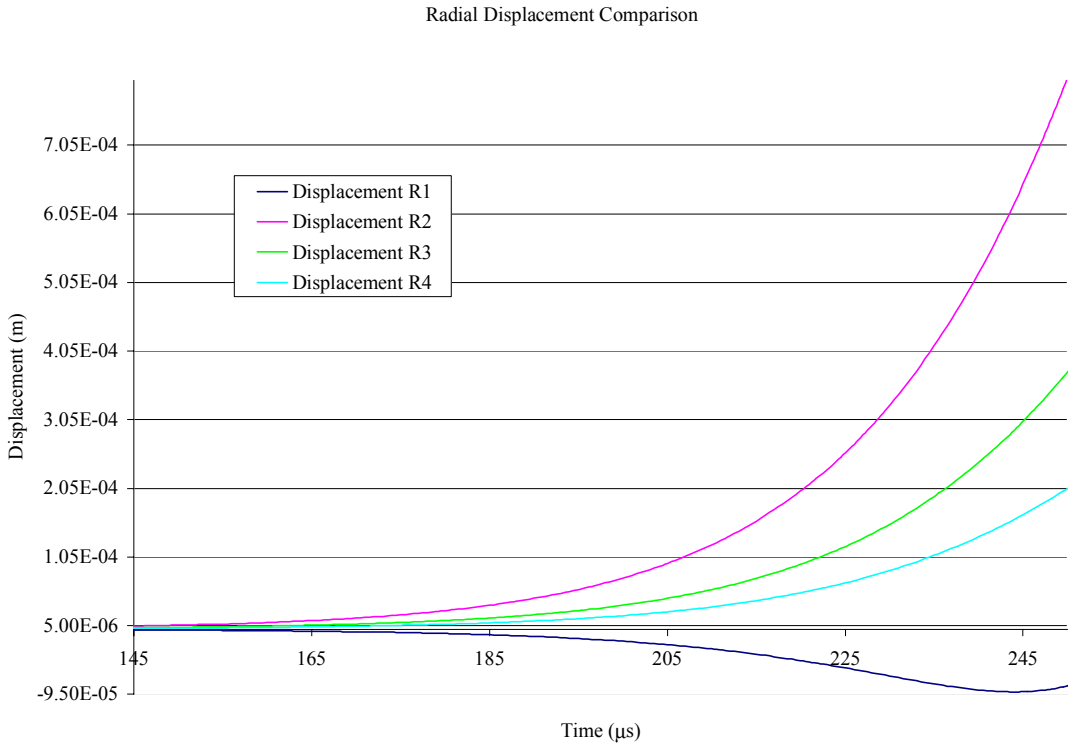


Figure 7.16: Radial displacement during a large (~13.5 cents prompt) addition

Once again, the interior face of the hollow cylinder shocks inward in the region of highest temperature change, represented by curve R1 in Figure 7.16. It is worth noting that the time position of the change in sign of the time derivative of the displacement curve corresponds closely with the maximum power in time. Only when the rate of thermal energy deposition begins to reduce does the material reverse its direction of expansion.

7.2.6 SPR III Spatial Flux Profile (Large Pulse)

The spatial profile of the neutron flux along the radial dimension for the large SPR III pulse is shown in Figure 7.17.

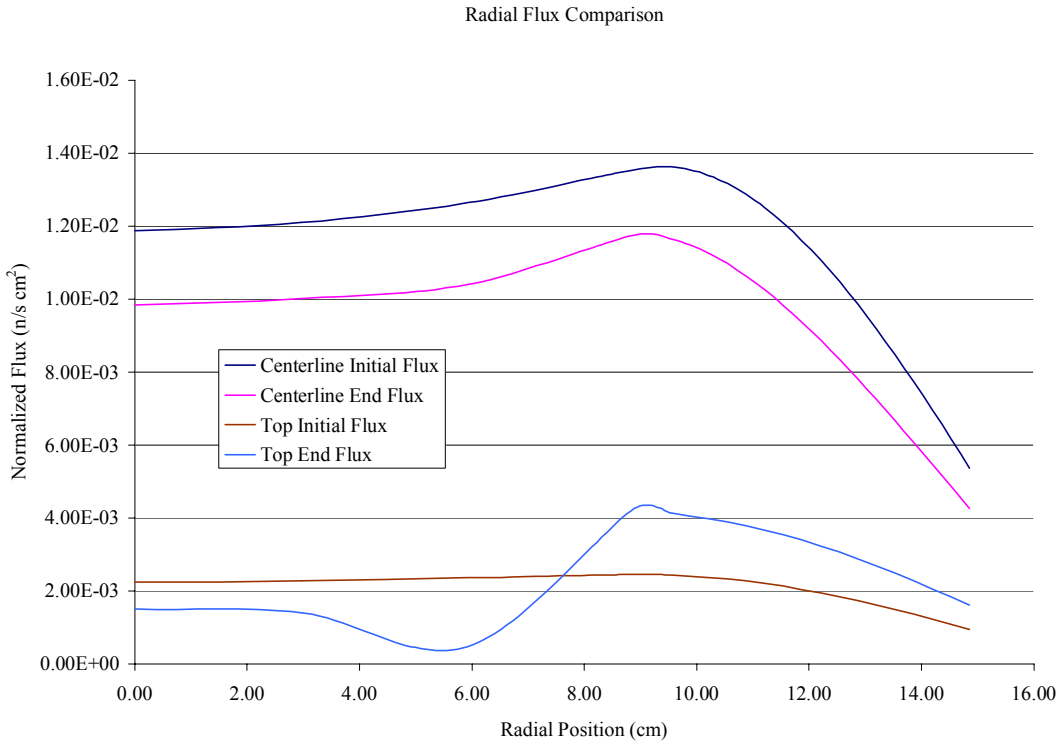


Figure 7.17: Spatial flux profile during a large (~13.5 cents prompt) addition

Once again, the flux behavior of the material interface located at 8.89 cm is modeled effectively by the finite element neutron diffusion solution. The flat initial condition evolves in time to reflect the significant flux drop in the hollow cavity as compared to the fuel mass.

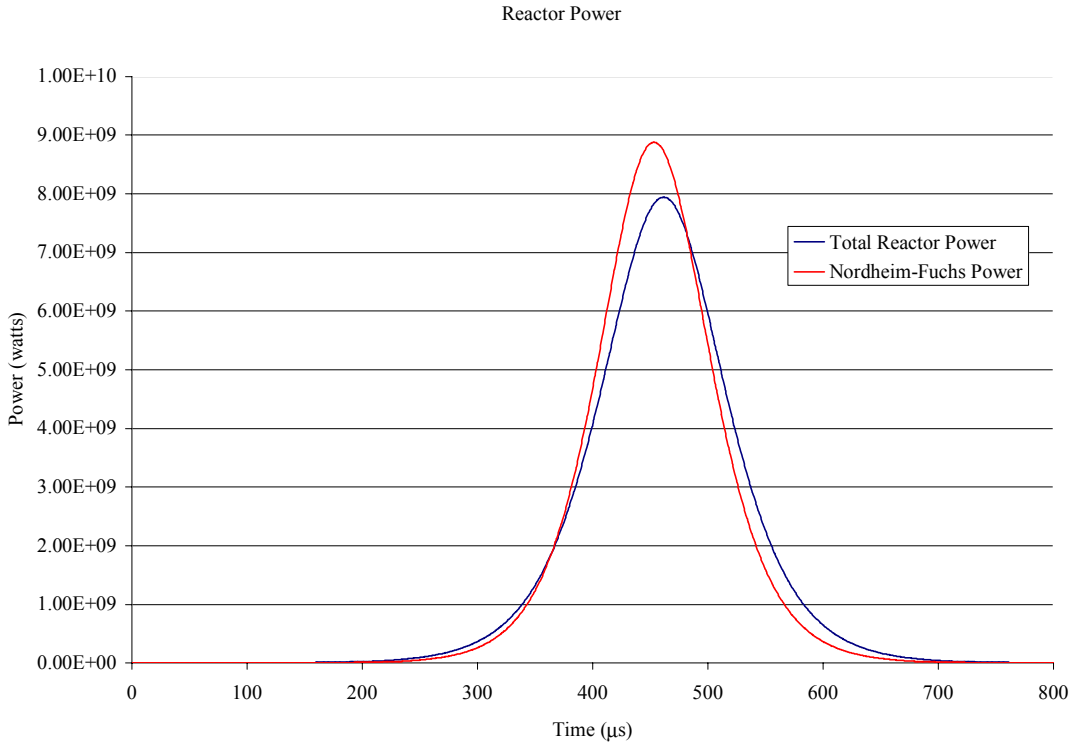
7.3 Spherical Reactor Results

Here similar results for a solid sphere of U-10 Mo are presented. The assembly is brought to a prompt supercritical state via the addition of fissile material to its surface. Again, a slow pulse and a fast pulse are presented to illustrate the disparity in their

characteristics. Estimated Nordheim-Fuchs parameters based off SPR values given in Table 2.2 are employed to provide another basis for comparison.

7.3.1 Spherical Power and Temperature Results (Small Pulse)

Power and temperature results for a prompt supercritical excursion are presented here for a solid spherical assembly. Figure 7.18 shows the power behavior in time produced by the one-dimensional code *ODMain* compared to a Nordheim-Fuchs fit estimated using the spherical reactor mass, a slightly shorter neutron lifetime than SPR II, and SPR shutdown characteristics in general. The approximate reactivity is \$1.04,



constituting a very small pulse.

Figure 7.18: Power comparison for a small (~4 cents prompt) addition

Figure 7.19 shows the corresponding temperature behavior in time for the power pulse illustrated above.

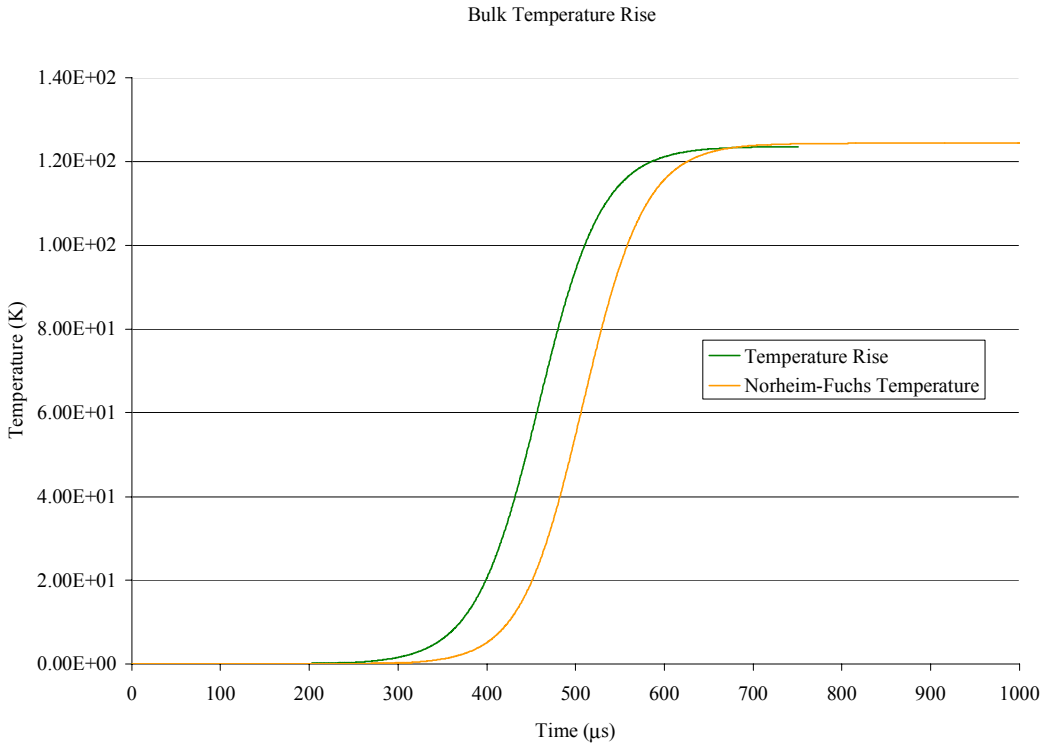


Figure 7.19: Temperature comparison for a small (~4 cents prompt) addition

7.3.2 Spherical Displacement Results (Small Pulse)

The displacement in an angularly symmetric sphere is obviously one-dimensional. In the solid case there is clearly no opportunity for a shock inward at an interior face. However, inertial lag still plays an important role in the shutdown behavior of this system. Figure 7.20 shows the radial expansion at a point near the core and a point on the sphere surface.

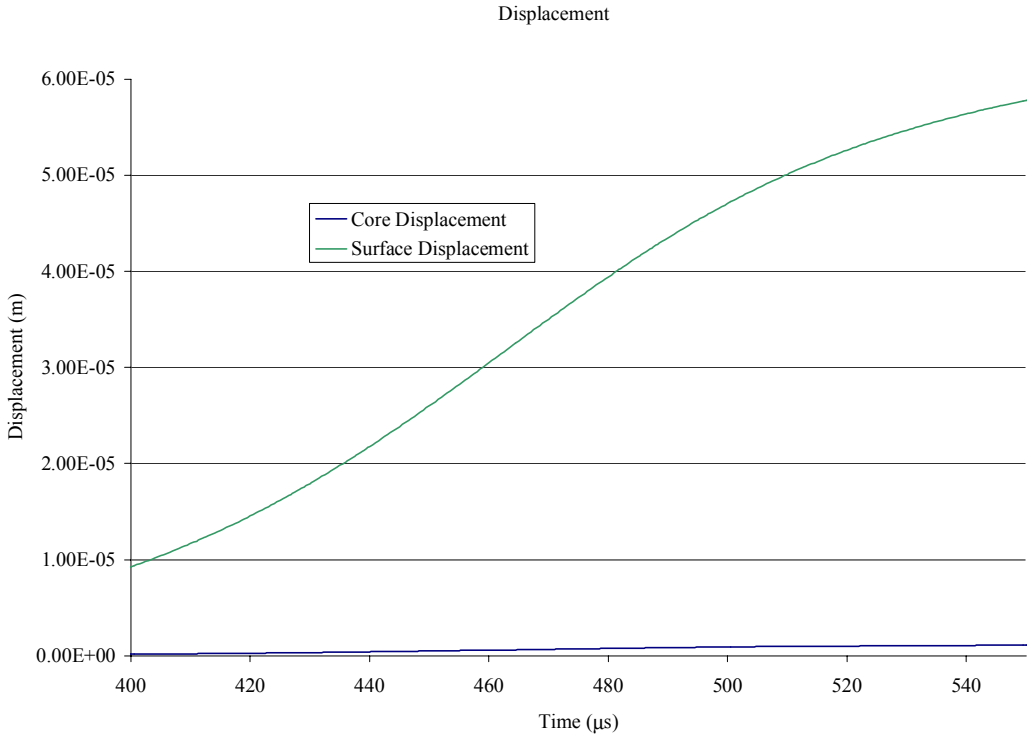


Figure 7.20: Displacement profile for a small (~4 cents prompt) addition

While inertial lag is not as pronounced in Figure 7.20 as it would be in a faster pulse, dynamic expansion must be simulated to properly reproduce reactor behavior.

7.3.3 Spherical Spatial Flux Profile (Small Pulse)

The radial flux profile during a slow pulse changes relatively little from the beginning of the pulse to the end. In fact, they are nearly indistinguishable plotted together, seen in Figure 7.21.

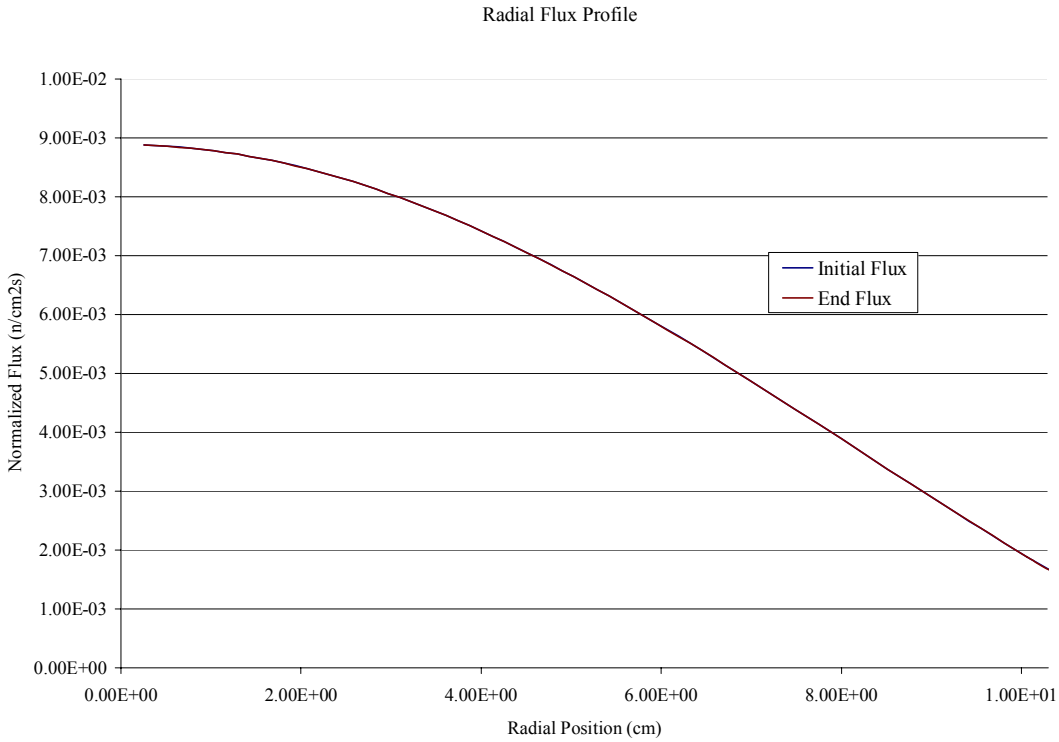


Figure 7.21: Radial flux profile for a small (~4 cents prompt) addition

The normalized flux profiled above is as expected for a solid spherical domain.

7.3.4 Spherical Power and Temperature Results (Large Pulse)

We present data comparisons for a pulse near imminent phase transition in its core. Figure 7.22 shows numerical data for a large spherical reactor pulse. We estimate the reactivity addition to be approximately $1.15 for this simulated pulse, and the data may be compared to that shown in Section 6.4.

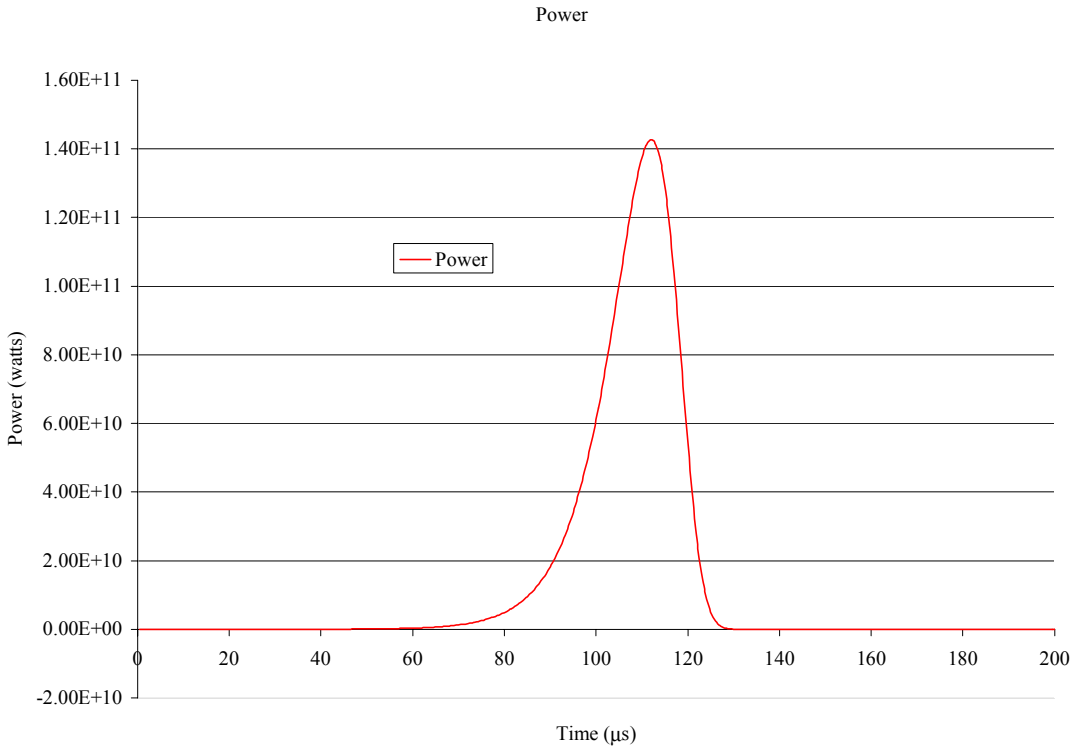


Figure 7.22: Power curve for a $1.15addition

The data used to generate the curve in Figure 7.22 were produced by code *ODMain*. This code performs more effectively than the MOL/MCNP method for this particular assembly, especially in reproducing the asymmetry of the power profile. Figure 7.23 shows the corresponding temperature behavior.

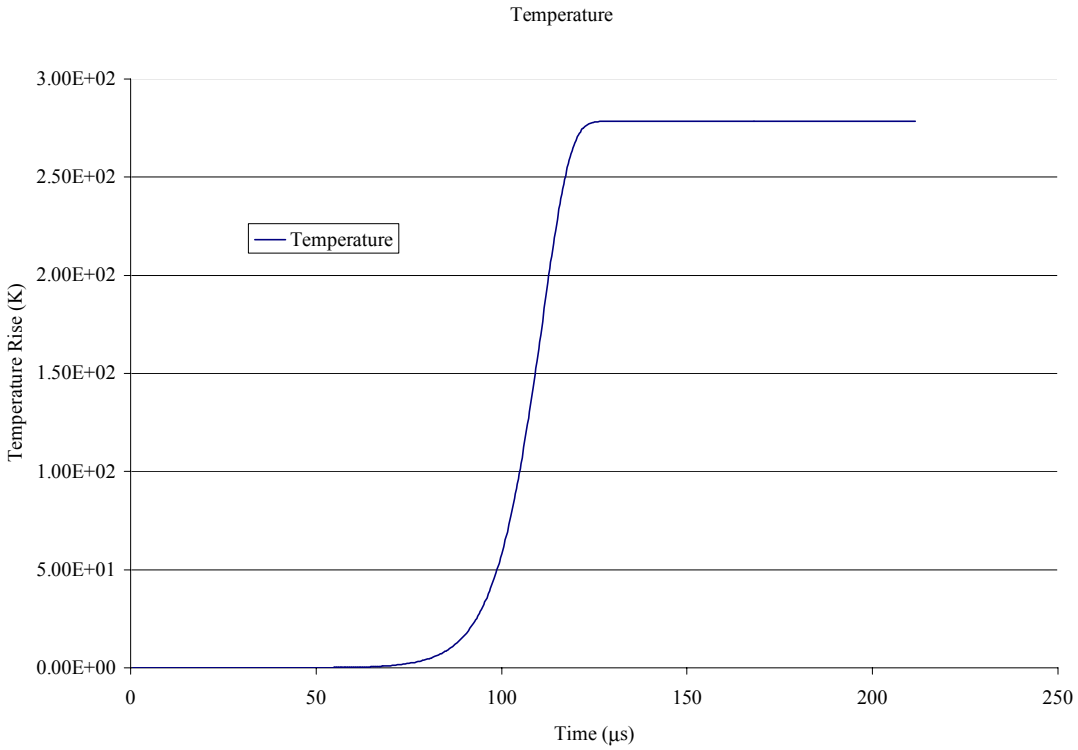


Figure 7.23: Temperature curve for a $1.15 addition

Figure 7.23 shows the temperature behavior as a function of time. It is worth noting the small time scale during which the temperature rise takes place. The corresponding displacement behavior is significantly delayed, as seen below.

7.3.5 Spherical Displacement Results (Large Pulse)

Figure 7.24 shows the radial displacement as a function of time for a large spherical pulse.

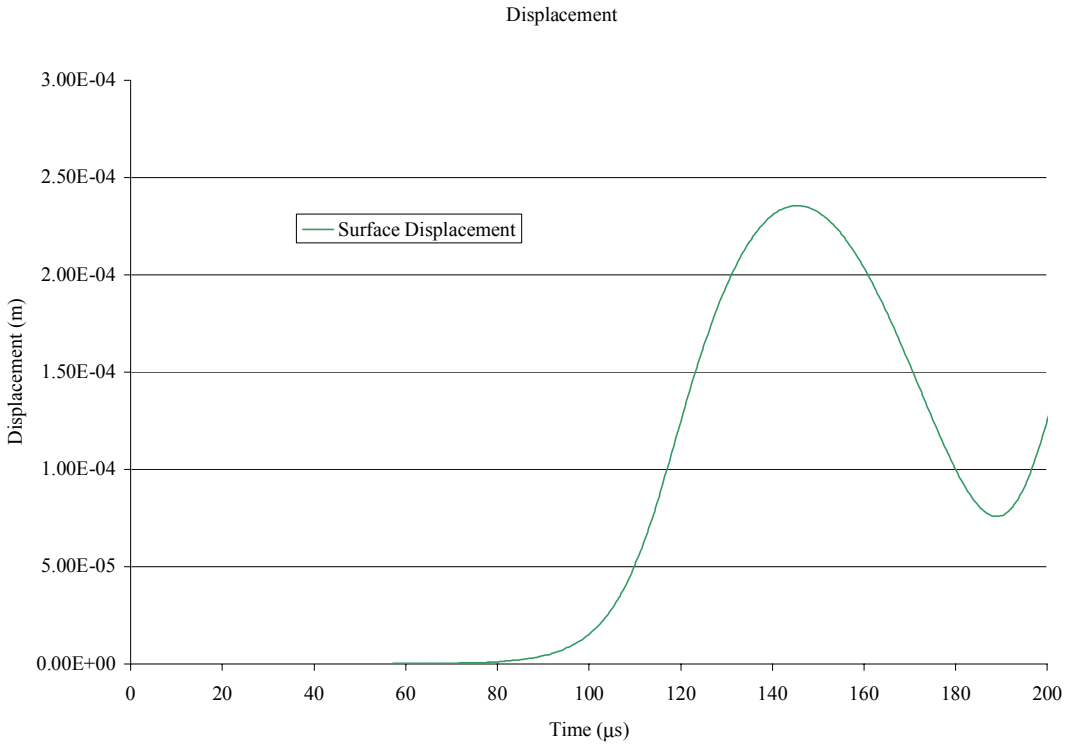


Figure 7.24: Radial displacement curve for a $1.15 addition

The displacement observed in Figure 7.24 is typical of a solid sphere expanding under a temperature transient. It is worth comparing the time behavior in Figure 7.24 and Figure 7.22: while the inertial lag in the smaller spherical pulse shown previously was difficult to observe, it is very evident in the larger pulse. By the time half the thermal energy has been deposited, the surface of the sphere has moved only about a fifth of the way to its equilibrium position. This demonstrates again that modeling this behavior is impossible without a dynamic elasticity solution.

7.3.6 Spherical Spatial Flux Profile (Large Pulse)

The spatial profile of the neutron flux along the radial dimension for a large pulse is shown in Figure 7.25.

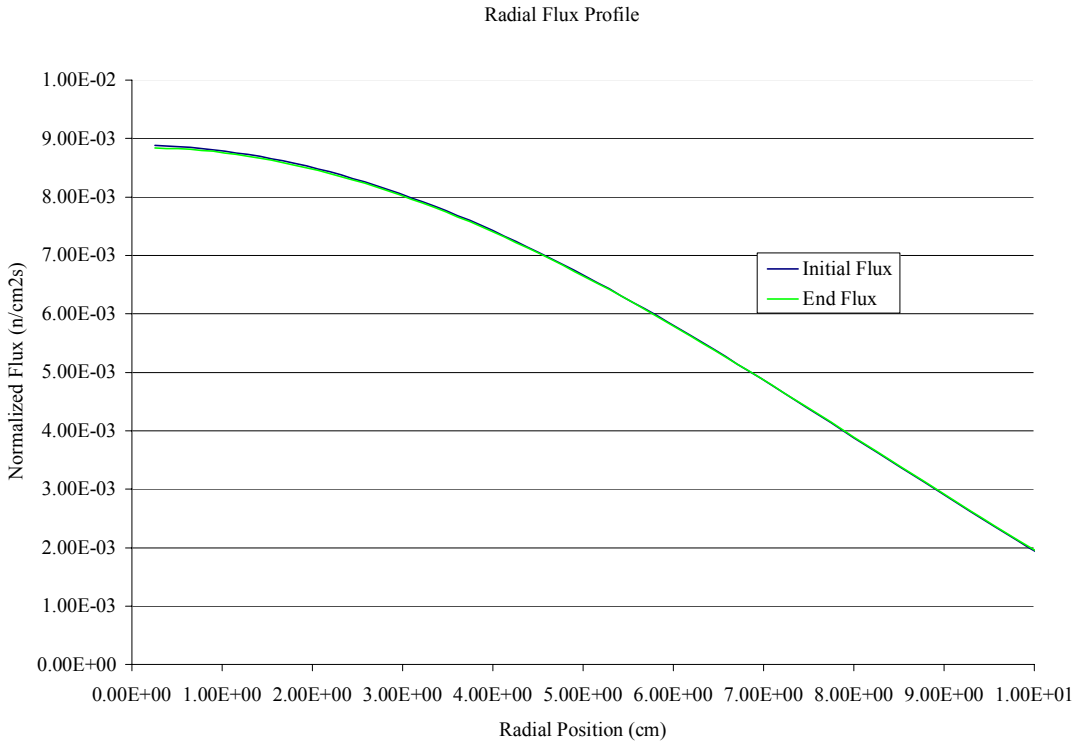


Figure 7.25: Spatial flux profile during a large (~15 cents prompt) addition

Once again, the changes in flux profiled from the beginning of the pulse to the end are difficult to differentiate in the spherical case. This is due to the relatively small dimensions of the reactor.

Chapter 8: Conclusions

8.1 Current Solution Strengths

The method-of-lines/MCNP solution method illustrated in Chapter 5 may be described as accurate since two significantly different pulses were produced that closely agree with observed SPR III data and the theoretical fits used to describe them. This validates the general solution algorithm employed in both this method and the finite element codes developed in this dissertation.

The finite element solution method exhibits several key strengths. First, the linear thermoelasticity model in one and two dimensions reproduced static results during preliminary testing without problems. Further, it closely reproduced the known and extrapolated dynamic expansion behavior of the SPR reactors. While it did not deliver an exact solution for the particular geometries of these reactors, it produced results close enough to reproduce the shutdown characteristics of these reactors to an excellent degree.

This point must be stressed. While neither the diffusion or elasticity solutions were perfect for the SPR geometries, the dynamic interaction between the two was accurate. Two real reactor systems' behaviors were closely modeled using several significant approximations. In both cases numerical data reproduced operational data. Further, the computational model reproduced asymmetric power behavior that cannot effectively be predicted using decoupled analyses. Curves may be fitted to these asymmetric power profiles and used to predict future behavior, but this requires operation of the reactor. The coupled dynamic solution implemented in the finite element codes developed in this dissertation produces this behavior from much more basic principles.

8.2 CURRENT SOLUTION WEAKNESSES

A variety of simplifying assumptions were made for the "patchwork" method, especially for the thermoelastic treatment. The eighteen fuel plates were treated as three bulk sections, and body forces and interfaces were entirely neglected. Since we sought only the criticality change, which occurs largely due to the radial displacement we solved directly for, we obtained good results. Compared to a complete thermomechanical treatment (Miller 1994), this solution is simplified to an extreme.

Since the SPR III geometry was built in a Monte Carlo code, we can accurately state that the neutron transport is more robust. A few simplifications to the geometry may introduce small errors. Since the kinetics and thermomechanics are coupled, however, any errors produced by one aspect of the solution necessarily propagate to the other.

Given the number of approximations made to obtain the results given in Chapter 6, one can accurately state that such a method is useful only in very limited applications.

The one-group neutron diffusion approximation to the neutron transport behavior stands out as the weakest point in the finite element model. This approximation was selected in part to avoid complicating the code needlessly in its current state. Another major consideration for using this approximation lay in the fact that there was no question it could be solved quickly on a single-processor machine. The effort required to develop these codes specifically to take advantage of multiple processors will be accomplished by future workers.

The time-dependent behavior of the diffusion approximation is far too dependent on the input parameters, especially the neutron velocity. When an accurate average velocity is selected, the results are good. When an inaccurate velocity is used, the power profile is either very flat or very sharp compared to the Nordheim-Fuchs approximation. This effect is noticeable in Figure 7.1.

Note however that regardless of the shape of the power profile, the system still tends to shut down correctly according to the dynamic interaction between thermal expansion and neutron diffusion.

8.3 FUTURE WORK

The neutron transport solution is the area most in need of refinement. While the dynamic solid model may be complicated ad infinitum to include various behaviors, its current implementation is accurate enough to model most reactors exhibiting dynamic fuel expansion effects. General anisotropy, plasticity and fracture prediction functionality would all contribute to the modeling of different reactor fuels, possibly in accident scenarios. The first and second of these would require relatively little effort to implement as long as suitable material data are available.

The neutron diffusion approximation implemented in this dissertation is only moderately suitable to model the behavior of highly enriched fast burst reactors such as SPR II and SPR III, as well as similar, theoretical nuclear assemblies. In order to predict the behavior of more complicated systems, a Monte Carlo solution in two or three dimensions should eventually replace the diffusion solution.

One potential route would involve the calculation of local multiplication constants on an arbitrary mesh. This in turn would allow use of the tried-and-true point kinetics equations in small spatial regions to handle time-dependence, in effect producing the accuracy of a full spatial kinetics solution but requiring less computational time. Further, this would permit the easy inclusion of delayed neutrons, resonance broadening and neutron reflection: all essential behaviors in more complicated systems.

8.4 FINAL COMMENTS

The goal of this dissertation project as stated in the initial proposal was "to develop and implement a finite element solution to the governing coupled thermoelastic and neutron kinetics equations for small, highly-enriched nuclear assemblies in spherical and cylindrical configurations."

This goal has been successfully accomplished. The coupled dynamic behaviors of these types of assemblies have been successfully reproduced by the codes developed in this dissertation.

Incorporation of time-dependent changes in material and load data is essential to modeling dynamic nuclear systems. A foundation for future work in the form of a tested solution method and program logic has been laid, and will hopefully result in more general codes analyzing broader classes of time-dependent nuclear systems.

Appendix A: Source Code for 1D Code

This is the complete Fortran 95 source code for the one-dimensional code solving the time-dependent neutron diffusion and thermoelasticity equations on a solid spherical domain. It compiles and runs successfully using the g95 Fortran compiler (www.g95.org).

Some text wraps occurred when moved to the word processor format, but these are obvious and can be avoided in reproducing the code.

```
!
!
!
!Software written by Stephen C. Wilson of the University of Texas
!
!This is a finite element code that solves the time-dependent
!coupled neutron diffusion and thermoelasticity equations
!on a spherical domain. See dissertation for complete
!documentation.
!
!Developed under contract with Sandia National Laboratories.
!
!June 28, 2006
!
!
!


MODULE ODGlobalConstants
IMPLICIT NONE



INTEGER, PARAMETER :: DBL = 8
INTEGER, PARAMETER :: NElem = 40
INTEGER, PARAMETER :: Ndof = NElem + 1
INTEGER, PARAMETER :: Order = 2
REAL(KIND=DBL), PARAMETER :: PI = 3.14159265358979
REAL(KIND=DBL), PARAMETER :: EF = 1.
REAL(KIND=DBL), PARAMETER :: Radius = 10.19 !+ 0.003
REAL(KIND=DBL), PARAMETER :: DensInit = 17.040
REAL(KIND=DBL), PARAMETER :: DensInitAir = 0.125
REAL(KIND=DBL), PARAMETER :: PhiOldInit = 1.D+10
REAL(KIND=DBL), PARAMETER :: EnergyPerFission = 320.4D-13
REAL(KIND=DBL), PARAMETER :: SourceTerm = 0.
REAL(KIND=DBL), PARAMETER :: Poisson = 0.38
REAL(KIND=DBL), PARAMETER :: NeutronVelocity = 1.D+9
REAL(KIND=DBL), PARAMETER :: FissionNubar = 2.6
```

```
REAL(KIND=DBL), PARAMETER :: sigmaF = 1.046D-24

REAL(KIND=DBL), PARAMETER :: sigmaA = 1.242D-24

REAL(KIND=DBL), PARAMETER :: sigmaAair = 0.06786D-24

REAL(KIND=DBL), PARAMETER :: sigmaS = 6.04D-24
REAL(KIND=DBL), PARAMETER :: sigmaSair = 2.438D-24
REAL(KIND=DBL), PARAMETER :: kel = 1.
REAL(KIND=DBL), PARAMETER :: A1DiffCoeff = 1.
REAL(KIND=DBL), PARAMETER :: F1DiffCoeff = 0.
REAL(KIND=DBL), PARAMETER :: A1ElasCoeff = 1.
INTEGER, PARAMETER :: NumTimeChunks = 1000

REAL(KIND=DBL), PARAMETER :: Time= 5.D-7

REAL(KIND=DBL), PARAMETER :: globalstep = 5.D-8

REAL(KIND=DBL), PARAMETER :: timestep_elas = globalstep
!REAL(KIND=DBL), PARAMETER :: alpha = 1./2.
REAL(KIND=DBL), PARAMETER :: Gamma = 1./2.

REAL(KIND=DBL), PARAMETER :: Beta = 1./4.

!REAL(KIND=DBL), PARAMETER :: elasA1 = alpha * timestep_elas

!REAL(KIND=DBL), PARAMETER :: elasA2 = (1. - alpha) * timestep_elas

!REAL(KIND=DBL), PARAMETER :: elasA3 = 1./(Beta * timestep_elas**2.)

!REAL(KIND=DBL), PARAMETER :: elasA4 = timestep_elas * elasA3

!REAL(KIND=DBL), PARAMETER :: elasA5 = (1. / Gamma) - 1.

!REAL(KIND=DBL), PARAMETER :: elasA6 = alpha / (Beta * timestep_elas)

!REAL(KIND=DBL), PARAMETER :: elasA7 = (alpha / Beta) - 1.

!REAL(KIND=DBL), PARAMETER :: elasA8 = ((alpha / Gamma) - 1.) * timestep_elas
REAL(KIND=DBL), PARAMETER :: DiffAlpha = 1./2.
REAL(KIND=DBL), PARAMETER :: diffB1 = DiffAlpha*globalstep
REAL(KIND=DBL), PARAMETER :: diffB2 = (1-DiffAlpha)*globalstep
REAL(KIND=DBL), PARAMETER :: DiffConstantInit = 1.12404


END MODULE ODGlobalConstants
!
!
!
MODULE ODFunctions1
IMPLICIT NONE

CONTAINS

        FUNCTION Xi(x,elem,Hsize)

        USE ODGlobalConstants

        IMPLICIT NONE

        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: x
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: Hsize
        REAL(KIND=DBL) :: Xi

        Xi = (2._DBL*x)/(Hsize(elem)) - 2._DBL*elem + 1
```

```
        END FUNCTION Xi

        !
        !

        FUNCTION B1ElasCoeff(x)

        USE ODGlobalConstants

        IMPLICIT NONE

        REAL(KIND=DBL), INTENT(IN) :: x
        REAL(KIND=DBL) :: B1ElasCoeff

        B1ElasCoeff = 2._DBL / (x**2._DBL)

        END FUNCTION B1ElasCoeff

END MODULE ODFunctions1
!
!
!
MODULE ODFunctions2
IMPLICIT NONE

CONTAINS

        FUNCTION Psi(elem,jay,x,Hsize)
        USE ODGlobalConstants
        USE ODFunctions1
        IMPLICIT NONE

        REAL(KIND=DBL), INTENT(IN) :: x
        INTEGER, INTENT(IN) :: elem, jay
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: Hsize
        REAL(KIND=DBL) :: Psi

        IF ( jay == 1 ) THEN
                Psi = (1._DBL/2._DBL) * (1._DBL - Xi(x,elem,Hsize))
        ELSE
                Psi = (1._DBL/2._DBL) * (1._DBL + Xi(x,elem,Hsize))
        END IF

        END FUNCTION Psi

        !
        !

        FUNCTION dPsidx(elem,jay,Hsize)
        USE ODGlobalConstants
        IMPLICIT NONE

        INTEGER,INTENT(IN) :: elem, jay
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: Hsize
        REAL(KIND=DBL) :: dPsidx

        IF ( jay == 1 ) THEN
                dPsidx = (-1._DBL)/(Hsize(elem))
        ELSE
                dPsidx = (1._DBL)/(Hsize(elem))
        END IF

        END FUNCTION dPsidx

END MODULE ODFunctions2
```

```
!
!
!

MODULE ODIntArgs
IMPLICIT NONE

CONTAINS

        SUBROUTINE KElasArg(x,elem,aye,jay,Hsize,InputCoeff,OutputScalar)
        USE ODFunctions1
        USE ODFunctions2
        USE ODGlobalConstants
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: elem,aye,jay
        REAL(KIND=DBL), INTENT(IN) :: x
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: Hsize,InputCoeff
        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

        REAL(KIND=DBL) :: tempscalar

        IF ( x /= 0. ) THEN
                tempscalar = InputCoeff(elem) * dPsidx(elem,aye,Hsize) *
dPsidx(elem,jay,Hsize) + B1ElasCoeff(x)* &

Psi(elem,aye,x,Hsize)*Psi(elem,jay,x,Hsize)
        ELSE
                tempscalar = InputCoeff(elem) * dPsidx(elem,aye,Hsize) *
dPsidx(elem,jay,Hsize)
        END IF

        OutputScalar = tempscalar

        END SUBROUTINE KElasArg

        !
        !

        SUBROUTINE MElasArg(x,elem,aye,jay,Hsize,InputCoeff,OutputScalar)
        USE ODFunctions1
        USE ODFunctions2
        USE ODGlobalConstants
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: elem,aye,jay
        REAL(KIND=DBL), INTENT(IN) :: x
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: Hsize,InputCoeff
        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

        REAL(KIND=DBL) :: tempscalar

        tempscalar = InputCoeff(elem) * Psi(elem,aye,x,Hsize) * Psi(elem,jay,x,Hsize)

        OutputScalar = tempscalar

        END SUBROUTINE MElasArg

        !
        !

        SUBROUTINE FElasArg(x,elem,aye,Hsize,InputCoeff,OutputScalar)
        USE ODFunctions1
        USE ODFunctions2
        USE ODGlobalConstants
        IMPLICIT NONE
```

```
INTEGER, INTENT(IN) :: elem,aye
REAL(KIND=DBL), INTENT(IN) :: x
REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: Hsize,InputCoeff
REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

REAL(KIND=DBL) :: tempscalar

tempscalar = InputCoeff(elem) * Psi(elem,aye,x,Hsize)

OutputScalar = tempscalar

END SUBROUTINE FElasArg

!
!

SUBROUTINE KDiffArg(x,elem,aye,jay,Hsize,InputCoeff,OutputScalar)
USE ODFunctions1
USE ODFunctions2
USE ODGlobalConstants
IMPLICIT NONE

INTEGER, INTENT(IN) :: elem,aye,jay
REAL(KIND=DBL), INTENT(IN) :: x
REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: Hsize,InputCoeff
REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

REAL(KIND=DBL) :: tempscalar

tempscalar = A1DiffCoeff * dPsidx(elem,aye,Hsize) * dPsidx(elem,jay,Hsize) + &
             InputCoeff(elem) * Psi(elem,aye,x,Hsize) * Psi(elem,jay,x,Hsize)

OutputScalar = tempscalar

END SUBROUTINE KDiffArg

!
!

SUBROUTINE MDiffArg(x,elem,aye,jay,Hsize,InputCoeff,OutputScalar)
USE ODFunctions1
USE ODFunctions2
USE ODGlobalConstants
IMPLICIT NONE

INTEGER, INTENT(IN) :: elem,aye,jay
REAL(KIND=DBL), INTENT(IN) :: x
REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: Hsize,InputCoeff
REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

REAL(KIND=DBL) :: tempscalar

tempscalar = InputCoeff(elem) * Psi(elem,aye,x,Hsize) * Psi(elem,jay,x,Hsize)

OutputScalar = tempscalar

END SUBROUTINE MDiffArg

!
!

SUBROUTINE FDiffArg(x,elem,aye,Hsize,InputCoeff,OutputScalar)
USE ODFunctions1
USE ODFunctions2
USE ODGlobalConstants
```

```
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: elem,aye
        REAL(KIND=DBL), INTENT(IN) :: x
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: Hsize,InputCoeff
        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

        REAL(KIND=DBL) :: tempscalar

        tempscalar = InputCoeff(elem) * Psi(elem,aye,x,Hsize)

        OutputScalar = tempscalar

        END SUBROUTINE FDiffArg

END MODULE ODIntArgs
!
!
!
MODULE ODIntRoutines

IMPLICIT NONE

CONTAINS

        SUBROUTINE Integrator1(func,NodePos,Hsize,InputCoeff,elem,aye,jay,OutputScalar)
        USE ODGlobalConstants
        IMPLICIT NONE

        EXTERNAL :: func ! subroutine called to generate function values at intpoints
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: InputCoeff ! vector of element
material data
        INTEGER, INTENT(IN) :: elem, aye, jay

        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

        REAL(KIND=DBL) :: IntCoeff, HX, Intsum, tempscalar
        INTEGER :: i !index
        REAL(KIND=DBL), DIMENSION(4) :: IntWeight, IntPoint

        Intsum = 0
        IntCoeff = 0.375_DBL
        HX = Hsize(elem) / 3._DBL

        DO i = 1,4
                IntPoint(i) = NodePos(elem) + HX*(i-1._DBL)
        END DO

        IntWeight(1) = 1._DBL
        IntWeight(2) = 3._DBL
        IntWeight(3) = IntWeight(2)
        IntWeight(4) = IntWeight(1)

        DO i = 1,4
                CALL func(IntPoint(i),elem,aye,jay,Hsize,InputCoeff,tempscalar)
                IntSum = IntSum + IntCoeff * HX * IntWeight(i) * tempscalar
        END DO

        OutputScalar = IntSum

        END SUBROUTINE Integrator1

        !
```

```
        !

        SUBROUTINE Integrator11(func,NodePos,Hsize,InputCoeff,elem,aye,jay,OutputScalar)
        USE ODGlobalConstants
        IMPLICIT NONE

        EXTERNAL :: func ! subroutine called to generate function values at intpoints
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof+1) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: InputCoeff ! vector of element
material data
        INTEGER, INTENT(IN) :: elem, aye, jay

        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

        REAL(KIND=DBL) :: IntCoeff, HX, Intsum, tempscalar
        INTEGER :: i !index
        REAL(KIND=DBL), DIMENSION(4) :: IntWeight, IntPoint

        Intsum = 0
        IntCoeff = 0.375_DBL
        HX = Hsize(elem) / 3._DBL

        DO i = 1,4
                IntPoint(i) = NodePos(elem) + HX*(i-1._DBL)
        END DO

        IntWeight(1) = 1._DBL
        IntWeight(2) = 3._DBL
        IntWeight(3) = IntWeight(2)
        IntWeight(4) = IntWeight(1)

        DO i = 1,4
                CALL func(IntPoint(i),elem,aye,jay,Hsize,InputCoeff,tempscalar)
                IntSum = IntSum + IntCoeff * HX * IntWeight(i) * tempscalar
        END DO

        OutputScalar = IntSum

        END SUBROUTINE Integrator11

        !
        !

        SUBROUTINE Integrator2(func,NodePos,Hsize,InputCoeff,elem,aye,OutputScalar)
        USE ODGlobalConstants
        IMPLICIT NONE

        EXTERNAL :: func ! subroutine called to generate function values at intpoints
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: InputCoeff ! vector of element
material data
        INTEGER, INTENT(IN) :: elem, aye

        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

        REAL(KIND=DBL) :: IntCoeff, HX, Intsum, tempscalar
        INTEGER :: i !index
        REAL(KIND=DBL), DIMENSION(4) :: IntWeight, IntPoint

        Intsum = 0
        IntCoeff = 0.375_DBL
        HX = Hsize(elem) / 3._DBL
```

```
        DO i = 1,4
                IntPoint(i) = NodePos(elem) + HX*(i-1._DBL)
        END DO

        IntWeight(1) = 1._DBL
        IntWeight(2) = 3._DBL
        IntWeight(3) = IntWeight(2)
        IntWeight(4) = IntWeight(1)

        DO i = 1,4
                CALL func(IntPoint(i),elem,aye,Hsize,InputCoeff,tempscalar)
                IntSum = IntSum + IntCoeff * HX * IntWeight(i) * tempscalar
        END DO

        OutputScalar = IntSum

        END SUBROUTINE Integrator2

        !
        !

        SUBROUTINE Integrator22(func,NodePos,Hsize,InputCoeff,elem,aye,OutputScalar)
        USE ODGlobalConstants
        IMPLICIT NONE

        EXTERNAL :: func ! subroutine called to generate function values at intpoints
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof+1) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: InputCoeff ! vector of element
material data
        INTEGER, INTENT(IN) :: elem, aye

        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

        REAL(KIND=DBL) :: IntCoeff, HX, Intsum, tempscalar
        INTEGER :: i !index
        REAL(KIND=DBL), DIMENSION(4) :: IntWeight, IntPoint

        Intsum = 0
        IntCoeff = 0.375_DBL
        HX = Hsize(elem) / 3._DBL

        DO i = 1,4
                IntPoint(i) = NodePos(elem) + HX*(i-1._DBL)
        END DO

        IntWeight(1) = 1._DBL
        IntWeight(2) = 3._DBL
        IntWeight(3) = IntWeight(2)
        IntWeight(4) = IntWeight(1)

        DO i = 1,4
                CALL func(IntPoint(i),elem,aye,Hsize,InputCoeff,tempscalar)
                IntSum = IntSum + IntCoeff * HX * IntWeight(i) * tempscalar
        END DO

        OutputScalar = IntSum

        END SUBROUTINE Integrator22

END MODULE ODIntRoutines

!
!
```

```
!

MODULE ODPreAssembly

IMPLICIT NONE

CONTAINS

        SUBROUTINE kdiffelem(elem,aye,jay,NodePos,Hsize,InputCoeff,OutputScalar)
        USE ODGlobalConstants
        USE ODIntRoutines
        USE ODIntArgs
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: elem,aye,jay
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof+1) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: InputCoeff ! vector of element
material data

        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

        REAL(KIND=DBL) :: tempscalar

        CALL Integrator11(KDiffArg,NodePos,Hsize,InputCoeff,elem,aye,jay,tempscalar)

        OutputScalar = tempscalar

        END SUBROUTINE kdiffelem

        !
        !

        SUBROUTINE mdiffelem(elem,aye,jay,NodePos,Hsize,InputCoeff,OutputScalar)
        USE ODGlobalConstants
        USE ODIntRoutines
        USE ODIntArgs
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: elem,aye,jay
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof+1) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: InputCoeff ! vector of element
material data

        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

        REAL(KIND=DBL) :: tempscalar

        CALL Integrator11(MDiffArg,NodePos,Hsize,InputCoeff,elem,aye,jay,tempscalar)

        OutputScalar = tempscalar

        END SUBROUTINE mdiffelem

        !
        !

        SUBROUTINE fdiffelem(elem,aye,NodePos,Hsize,InputCoeff,OutputScalar)
        USE ODGlobalConstants
        USE ODIntRoutines
        USE ODIntArgs
        IMPLICIT NONE
```

```
        INTEGER, INTENT(IN) :: elem,aye
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof+1) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: InputCoeff ! vector of element
material data

        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

        REAL(KIND=DBL) :: tempscalar

        CALL Integrator22(FDiffArg,NodePos,Hsize,InputCoeff,elem,aye,tempscalar)

        OutputScalar = tempscalar

        END SUBROUTINE fdiffelem

        !
        !

        SUBROUTINE kelaselem(elem,aye,jay,NodePos,Hsize,InputCoeff,OutputScalar)
        USE ODGlobalConstants
        USE ODIntRoutines
        USE ODIntArgs
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: elem,aye,jay
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: InputCoeff ! vector of element
material data

        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

        REAL(KIND=DBL) :: tempscalar

        CALL Integrator1(KElasArg,NodePos,Hsize,InputCoeff,elem,aye,jay,tempscalar)

        OutputScalar = tempscalar

        END SUBROUTINE kelaselem

        !
        !

        SUBROUTINE melaselem(elem,aye,jay,NodePos,Hsize,InputCoeff,OutputScalar)
        USE ODGlobalConstants
        USE ODIntRoutines
        USE ODIntArgs
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: elem,aye,jay
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: InputCoeff ! vector of element
material data

        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

        REAL(KIND=DBL) :: tempscalar

        CALL Integrator1(MElasArg,NodePos,Hsize,InputCoeff,elem,aye,jay,tempscalar)

        OutputScalar = tempscalar
```

```
        END SUBROUTINE melaselem

        !
        !

        SUBROUTINE felaselem(elem,aye,NodePos,Hsize,InputCoeff,OutputScalar)
        USE ODGlobalConstants
        USE ODIntRoutines
        USE ODIntArgs
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: elem,aye
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: InputCoeff ! vector of element
material data

        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar

        REAL(KIND=DBL) :: tempscalar

        CALL Integrator2(FElasArg,NodePos,Hsize,InputCoeff,elem,aye,tempscalar)

        OutputScalar = tempscalar

        END SUBROUTINE felaselem

        !
        !

END MODULE ODPreAssembly

!
!
!

MODULE ODAssemblyRoutines

IMPLICIT NONE

CONTAINS

        SUBROUTINE MatrixAssemble1(func,NodePos,Hsize,InputCoeff,OutputMatrix)
        USE ODGlobalConstants

        IMPLICIT NONE

        EXTERNAL :: func ! calls subroutine to generate stiffness matrix entries for
assembly
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: InputCoeff ! vector of element
material data

        REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof,Ndof) :: OutputMatrix

        REAL(KIND=DBL), DIMENSION(Ndof,Ndof) :: tempmatrix1, tempmatrix2
        REAL(KIND=DBL) :: tempscalar
        INTEGER :: e,i,j,r,s ! indices

        tempmatrix1 = 0._DBL
        tempmatrix2 = 0._DBL
```

```
        DO e = 1,Nelem
                DO i = 1,2
                        DO j = 1,2
                                CALL func(e,i,j,NodePos,Hsize,InputCoeff,tempscalar)
                                IF ( (i+e-1) >= (j+e-1) ) THEN
                                        tempmatrix1((i+e-1),(j+e-1)) = tempmatrix1((i+e-
1),(j+e-1)) + tempscalar
                                ELSE
                                        tempmatrix1((i+e-1),(j+e-1)) = 0._DBL
                                END IF
                        END DO
                END DO
        END DO

        DO r = 1,Ndof
                DO s = 1,Ndof
                        IF ( s > r ) THEN
                                tempmatrix2(r,s) = tempmatrix1(s,r)
                        ELSE
                                tempmatrix2(r,s) = tempmatrix1(r,s)
                        END IF
                END DO
        END DO

        OutputMatrix = tempmatrix2

        END SUBROUTINE MatrixAssemble1

        !
        !

        SUBROUTINE MatrixAssemble2(func,NodePos,Hsize,InputCoeff,OutputMatrix)
        USE ODGlobalConstants

        IMPLICIT NONE

        EXTERNAL :: func ! calls subroutine to generate stiffness matrix entries for
assembly
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof+1) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: InputCoeff ! vector of element
material data

        REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof+1,Ndof+1) :: OutputMatrix

        REAL(KIND=DBL), DIMENSION(Ndof+1,Ndof+1) :: tempmatrix1, tempmatrix2
        REAL(KIND=DBL) :: tempscalar
        INTEGER :: e,i,j,r,s ! indices

        tempmatrix1 = 0._DBL
        tempmatrix2 = 0._DBL

        DO e = 1,Nelem+1
                DO i = 1,2
                        DO j = 1,2
                                CALL func(e,i,j,NodePos,Hsize,InputCoeff,tempscalar)
                                IF ( (i+e-1) >= (j+e-1) ) THEN
                                        tempmatrix1((i+e-1),(j+e-1)) = tempmatrix1((i+e-
1),(j+e-1)) + tempscalar
                                ELSE
                                        tempmatrix1((i+e-1),(j+e-1)) = 0._DBL
                                END IF
                        END DO
                END DO
        END DO
```

```
        DO r = 1,Ndof+1
                DO s = 1,Ndof+1
                        IF ( s > r ) THEN
                                tempmatrix2(r,s) = tempmatrix1(s,r)
                        ELSE
                                tempmatrix2(r,s) = tempmatrix1(r,s)
                        END IF
                END DO
        END DO

        OutputMatrix = tempmatrix2

        END SUBROUTINE MatrixAssemble2

        !
        !

        SUBROUTINE VectorAssemble1(func,NodePos,Hsize,InputCoeff,OutputVector)
        USE ODGlobalConstants

        IMPLICIT NONE

        EXTERNAL :: func ! calls subroutine to generate stiffness matrix entries for
assembly
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem) :: InputCoeff ! vector of element
material data

        REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof) :: OutputVector

        REAL(KIND=DBL), DIMENSION(Ndof) :: tempvector
        REAL(KIND=DBL) :: tempscalar
        INTEGER :: e,i ! indices

        tempvector = 0._DBL

        DO e = 1,Nelem
                DO i = 1,2
                        CALL func(e,i,NodePos,Hsize,InputCoeff,tempscalar)
                        tempvector(i+e-1) = tempvector(i+e-1) + tempscalar
                END DO
        END DO

        OutputVector = tempvector

        END SUBROUTINE VectorAssemble1

        !
        !

        SUBROUTINE VectorAssemble2(func,NodePos,Hsize,InputCoeff,OutputVector)
        USE ODGlobalConstants

        IMPLICIT NONE

        EXTERNAL :: func ! calls subroutine to generate stiffness matrix entries for
assembly
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof+1) :: NodePos ! vector of nodal
coordinates
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: Hsize ! vector of element sizes
        REAL(KIND=DBL), INTENT(IN), DIMENSION(NElem+1) :: InputCoeff ! vector of element
material data
```

```
REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof+1) :: OutputVector

REAL(KIND=DBL), DIMENSION(Ndof+1) :: tempvector
REAL(KIND=DBL) :: tempscalar
INTEGER :: e,i ! indices

tempvector = 0._DBL

DO e = 1,Nelem+1
        DO i = 1,2
                CALL func(e,i,NodePos,Hsize,InputCoeff,tempscalar)
                tempvector(i+e-1) = tempvector(i+e-1) + tempscalar
        END DO
END DO

OutputVector = tempvector

END SUBROUTINE VectorAssemble2

!
!

SUBROUTINE MatrixTruncate1(InputMatrix,OutputMatrix)
USE ODGlobalConstants

IMPLICIT NONE

REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof+1,Ndof+1) :: InputMatrix

REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof-1,Ndof-1) :: OutputMatrix

INTEGER :: q,r ! indices

DO q = 1,Ndof-1
        DO r = 1,Ndof-1
                OutputMatrix(q,r) = InputMatrix(q+1,r+1)
        END DO
END DO

END SUBROUTINE MatrixTruncate1

!
!

SUBROUTINE MatrixTruncate2(InputMatrix,OutputMatrix)
USE ODGlobalConstants

IMPLICIT NONE

REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof,Ndof) :: InputMatrix

REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof-1,Ndof-1) :: OutputMatrix

INTEGER :: q,r ! indices

DO q = 1,Ndof-1
        DO r = 1,Ndof-1
                OutputMatrix(q,r) = InputMatrix(q+1,r+1)
        END DO
END DO

END SUBROUTINE MatrixTruncate2

!
!
```

```
        SUBROUTINE VectorTruncate1(InputVector,OutputVector)
        USE ODGlobalConstants

        IMPLICIT NONE

        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof+1) :: InputVector

        REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof-1) :: OutputVector

        INTEGER :: q ! indices

        DO q = 1,Ndof-1
                OutputVector(q) = InputVector(q+1)
        END DO

        END SUBROUTINE VectorTruncate1

        !
        !

        SUBROUTINE VectorTruncate2(InputVector,OutputVector)
        USE ODGlobalConstants

        IMPLICIT NONE

        REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof) :: InputVector

        REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof-1) :: OutputVector

        INTEGER :: q ! indices

        DO q = 1,Ndof-1
                OutputVector(q) = InputVector(q+1)
        END DO

        END SUBROUTINE VectorTruncate2

        !
        !

        SUBROUTINE MatrixLump(SquareMatrix,dimMatrix,OutputMatrix)
  USE ODGlobalConstants
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN) :: dimMatrix ! dimension of square matrix to be row-lumped
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimMatrix,dimMatrix) :: SquareMatrix ! matrix to
be row-lumped
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(dimMatrix,dimMatrix) :: OutputMatrix ! row-
lumped matrix
  !
  REAL(KIND=DBL), DIMENSION(dimMatrix) :: RowSum ! sums row entries, condenses to this
vector
  REAL(KIND=DBL), DIMENSION(dimMatrix,dimMatrix) :: P ! tempmatrix
  INTEGER :: i,j ! indices
  !
  DO i = 1,dimMatrix
        RowSum(i) = 0
  END DO
  DO i = 1,dimMatrix
        DO j = 1,dimMatrix
                P(i,j) = 0
        END DO
  END DO
  DO i = 1,dimMatrix
        DO j = 1,dimMatrix
                RowSum(i) = RowSum(i) + SquareMatrix(i,j)
```

```
        END DO
  END DO
  DO i = 1,dimMatrix
        P(i,i) = RowSum(i)
  END DO
  OutputMatrix = P
  END SUBROUTINE MatrixLump
  !
  !
  !

END MODULE ODAssemblyRoutines

!
!
!

MODULE ODSolverRoutines1
IMPLICIT NONE


CONTAINS
        SUBROUTINE Augmented(InputMatrix,InputVector,ndim,OutputMatrix)
        USE ODGlobalConstants
        IMPLICIT NONE
        INTEGER, INTENT(IN) :: ndim
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim) :: InputMatrix
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim) :: InputVector
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(ndim,ndim+1) :: OutputMatrix

        INTEGER :: i,j ! indices

        OutputMatrix = 0._DBL

        DO i = 1,ndim
                DO j = 1,ndim
                        OutputMatrix(i,j) = InputMatrix(i,j)
                END DO
        END DO

        DO i = 1,ndim
                OutputMatrix(i,ndim+1) = InputVector(i)
        END DO

        END SUBROUTINE Augmented

        !
        !

        SUBROUTINE RowSwitch(InputMatrix,ndim,Row1,Row2,OutputMatrix)
        USE ODGlobalConstants
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: ndim
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim+1) :: InputMatrix
        INTEGER, INTENT(IN) :: Row1,Row2

        REAL(KIND=DBL), INTENT(OUT), DIMENSION(ndim,ndim+1) :: OutputMatrix

        REAL(KIND=DBL), DIMENSION(ndim,ndim+1) :: tempmatrix
        REAL(KIND=DBL), DIMENSION(ndim+1) :: temprow1, temprow2
        INTEGER :: i,j,Nrows,Ncols

        Nrows = ndim
        Ncols = ndim+1
```

```
tempmatrix = InputMatrix

DO j = 1,Ncols
       temprow1(j) = InputMatrix(Row1,j)
END DO

DO j = 1,Ncols
       temprow2(j) = InputMatrix(Row2,j)
END DO

DO j = 1,Ncols
       tempmatrix(Row1,j) = temprow2(j)
       tempmatrix(Row2,j) = temprow1(j)
END DO

OutputMatrix = tempmatrix

END SUBROUTINE RowSwitch

!
!

SUBROUTINE GetDivFactor(InputMatrix,ndim,Pivot,ColNum,OutputVector)
USE ODGlobalConstants
IMPLICIT NONE

INTEGER, INTENT(IN) :: ndim
REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim+1) :: InputMatrix
INTEGER, INTENT(IN) :: Pivot,ColNum
REAL(KIND=DBL), INTENT(OUT), DIMENSION(ndim) :: OutputVector

REAL(KIND=DBL) :: FactorElement
INTEGER :: i,j

FactorElement = InputMatrix(Pivot,ColNum)

DO i = Pivot,ndim
       OutputVector(i) = InputMatrix(i,ColNum) / FactorElement
END DO

END SUBROUTINE GetDivFactor

!
!

SUBROUTINE SearchForPivot(InputMatrix,ndim,RowStart,ColNum,OutputInteger)
USE ODGlobalConstants
IMPLICIT NONE

INTEGER, INTENT(IN) :: ndim,RowStart,ColNum
REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim+1) :: InputMatrix
INTEGER, INTENT(OUT) :: OutputInteger

INTEGER :: pivot,i

pivot = 1

DO i = RowStart,ndim
       IF ( InputMatrix(i,ColNum) /= 0. ) THEN
              pivot = i
              EXIT
       END IF
END DO

OutputInteger = pivot
```

```
        END SUBROUTINE SearchForPivot


END MODULE ODSolverRoutines1

MODULE ODSolverRoutines2

IMPLICIT NONE

CONTAINS

        SUBROUTINE ForwardElim(InputMatrix,ndim,OutputMatrix)
        USE ODGlobalConstants
        USE ODSolverRoutines1
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: ndim
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim+1) :: InputMatrix
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(ndim,ndim+1) :: OutputMatrix

        INTEGER :: i,j,Nrows,Ncols,rowstart,PivotRow,istat,status
        REAL(KIND=DBL), DIMENSION(ndim,ndim+1) :: tempmatrixB,tempmatrixQ
        REAL(KIND=DBL) :: FactorElement
        REAL(KIND=DBL), DIMENSION(ndim) :: DivFactor

        Nrows = ndim
        Ncols = ndim+1
        rowstart = 1

        PivotRow = 1

        tempmatrixB = InputMatrix
        tempmatrixQ = InputMatrix

        DO
                IF ( rowstart > Nrows ) EXIT
                tempmatrixQ = tempmatrixB
                CALL SearchForPivot(tempmatrixB,ndim,rowstart,rowstart,PivotRow)
                IF ( PivotRow /= rowstart ) THEN
                        CALL RowSwitch(tempmatrixB,ndim,PivotRow,rowstart,tempmatrixB)
                        CALL RowSwitch(tempmatrixQ,ndim,PivotRow,rowstart,tempmatrixQ)
                        PivotRow = rowstart
                END IF
                FactorElement = tempmatrixB(PivotRow,rowstart)
                CALL GetDivFactor(tempmatrixB,ndim,PivotRow,rowstart,DivFactor)
                DO i = rowstart,Nrows
                        DO j = rowstart,Ncols
                                IF ( i /= PivotRow ) THEN
                                        tempmatrixB(i,j) = tempmatrixQ(i,j) -
DivFactor(i)*tempmatrixQ(PivotRow,j)
                                ELSE
                                        tempmatrixB(i,j) = tempmatrixQ(i,j) / FactorElement
                                END IF
                        END DO
                END DO
                rowstart = rowstart + 1
        END DO

        OutputMatrix = tempmatrixB

        END SUBROUTINE ForwardElim

        !
        !

        SUBROUTINE BackSub(InputMatrix,ndim,OutputMatrix)
```

```
        USE ODGlobalConstants
        USE ODSolverRoutines1
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: ndim
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim+1) :: InputMatrix
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(ndim,ndim+1) :: OutputMatrix

        INTEGER :: i,j,Nrows,Ncols,rowstart,colstart,istat
        REAL(KIND=DBL), DIMENSION(ndim,ndim+1) :: tempmatrixB
        REAL(KIND=DBL) :: DivFact

        Nrows = ndim
        Ncols = ndim+1

        tempmatrixB = InputMatrix

        rowstart = Nrows

        DO
                IF ( rowstart <= 1 ) EXIT
                        DO i = 1,(rowstart-1)
                                DivFact = tempmatrixB(i,rowstart) /
tempmatrixB(rowstart,rowstart)
                                DO j = 1,Ncols
                                        tempmatrixB(i,j) = tempmatrixB(i,j) -
tempmatrixB(rowstart,j)*DivFact
                                END DO
                        END DO
                rowstart = rowstart - 1
        END DO

        OutputMatrix = tempmatrixB

        END SUBROUTINE BackSub

END MODULE ODSolverRoutines2

!
!
!

MODULE ODGaussSolver
IMPLICIT NONE

CONTAINS

        SUBROUTINE GJSolver(InputMatrix,InputVector,ndim,OutputVector)
        USE ODGlobalConstants
        USE ODSolverRoutines1
        USE ODSolverRoutines2
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: ndim
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim) :: InputMatrix
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim) :: InputVector
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(ndim) :: OutputVector

        REAL(KIND=DBL), DIMENSION(ndim,ndim+1) :: tempmatrix

        INTEGER :: i

        CALL Augmented(InputMatrix,InputVector,ndim,tempmatrix)
        CALL ForwardElim(tempmatrix,ndim,tempmatrix)
        CALL BackSub(tempmatrix,ndim,tempmatrix)
```

```
        DO i = 1, ndim
                OutputVector(i) = tempmatrix(i,ndim+1)
        END DO

        END SUBROUTINE GJSolver

END MODULE ODGaussSolver
!
!
!

PROGRAM ODMain

USE ODGlobalConstants
USE ODFunctions1
USE ODFunctions2
USE ODIntArgs
USE ODIntRoutines
USE ODPreAssembly
USE ODGaussSolver
USE ODSolverRoutines1
USE ODSolverRoutines2
USE ODAssemblyRoutines

IMPLICIT NONE

! variables for main program block
REAL(KIND=DBL) :: t,step ! time counter for inner while loops, step size thereof
INTEGER :: i,j,q,r,s,qq ! indices
INTEGER :: TimeChunks ! counter for main program while loops
REAL(KIND=DBL), DIMENSION(Ndof) :: ElasNodePos,QNeumann
REAL(KIND=DBL), DIMENSION(Ndof+1) :: DiffNodePos
INTEGER :: diffsolverdim, elassolverdim
REAL(KIND=DBL), DIMENSION(Ndof-1) :: Phiold ! IC for diffusion
REAL(KIND=DBL), DIMENSION(NElem) :: Disp,Vel,Acc,AVector ! vectors used in elas
differencing
REAL(KIND=DBL), DIMENSION(NElem) :: Disppred,Velpred,Dispnext,Velnext,Accnext
REAL(KIND=DBL), DIMENSION(NElem) :: Dispprev,Velprev,Accprev,Aprev
REAL(KIND=DBL), DIMENSION(NElem) :: Temp, LinearExpansion, SpecificHeat,DTempDR
REAL(KIND=DBL), DIMENSION(NElem) ::
RefTemp,YoungsModulus,Power,A1Elas,C1Elas,F1Elas,ElasHelem
REAL(KIND=DBL), DIMENSION(NElem+1) ::
Ncc,SigmaFission,SigmaAbsorption,DiffusionLength,NeutronSource,SigmaScatter
REAL(KIND=DBL), DIMENSION(NElem+1) ::
C1Diff,B1Diff,ElemCenter,DiffHelem,Dens,Volume,Mass,F1Diff
REAL(KIND=DBL), DIMENSION(NElem+1) :: AtomicWeight,AtomicNumber
REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:,:) :: KDiff,MDiff
REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:) :: FDiff
REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:,:) :: KDiffFinal,MDiffFinal, MLoad
REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:) :: FDiffFinal, FLoad
REAL(KIND=DBL), DIMENSION(Ndof-1) :: PhiNew
REAL(KIND=DBL), DIMENSION(Ndof+1) :: PhiNodal
REAL(KIND=DBL), DIMENSION(NElem) :: PhiElem,WaveVelocity
REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:,:) :: KElas, MElas,
KElasFinal,MElasFinal,KElasHat,MElasLoad
REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:) :: FElas, FElasFinal, FElasLoad
REAL(KIND=DBL), DIMENSION(Ndof) :: ElasDisplacement
REAL(KIND=DBL), DIMENSION(Ndof) :: DiffDisplacement
REAL(KIND=DBL) :: tempscalar2 ! holder for accumulating volume vectors
REAL(KIND=DBL) :: alphael,betael,gammael
INTEGER :: ierror,istat,status
!
! call value setting routines to initialize some of the above arrays
! initializing other arrays
Phiold = PhiOldInit
!Disp = 0._DBL
```

```
!Vel = 0._DBL
!Acc = 0._DBL
!AVector = 0._DBL

Dispprev = 0._DBL
Velprev = 0._DBL
Accprev = 0._DBL

!Aprev = 0._DBL

Disppred = 0._DBL
Velpred = 0._DBL
Dispnext = 0._DBL
Velnext = 0._DBL
Accnext = 0._DBL

DO q = 1,Ndof
        DiffNodePos(q) = Radius * (q-1._DBL) / NElem
        ElasNodePos(q) = (Radius/100._DBL) * (q-1._DBL) / NElem
END DO

DiffNodePos(Ndof+1) = DiffNodePos(Ndof) + 2.13*DiffConstantInit


Temp = 25._DBL
Power = 0._DBL
LinearExpansion = Temp*9.786468D-9 + 9.918253D-6
SpecificHeat = Temp*0.118526_DBL + 104.69832_DBL
DTempDR = 0._DBL

DO q = 1,NElem+1
        IF ( q < NElem+1 ) THEN
                Dens(q) = DensInit
        ELSE
                Dens(q) = DensInitAir
        END IF
END DO


! set initial volume vector
DO i = 1,NElem+1
        Volume(i) = (4._DBL*PI/3._DBL) * ( (DiffNodePos(i+1))**3._DBL -
(DiffNodePos(i))**3._DBL )
END DO

DO i = 1, NElem+1
        Mass(i) = Dens(i)*Volume(i)
END DO



! conditional statement needed to set atomicweight and atomicnumber
DO i = 1,NElem+1
        IF ( i < NElem+1 ) THEN
                AtomicWeight(i) = 207.28789_DBL
                AtomicNumber(i) = 82._DBL
        ELSE
                AtomicWeight(i) = 14.46_DBL
                AtomicNumber(i) = 7.23_DBL
        END IF
END DO

Ncc = 0._DBL
SigmaFission = 0._DBL
SigmaAbsorption = 0._DBL
DiffusionLength = 0._DBL
```

```
NeutronSource = SourceTerm ! in global constants
C1Diff = 0._DBL
B1Diff = 0._DBL
PhiNodal = 0._DBL
RefTemp = 0._DBL
YoungsModulus = -1.057457D+8 * Temp + 9.5445D+10
DTempDR = 0._DBL
ElasDisplacement = 0._DBL
DiffDisplacement = 0._DBL
!
! open files for writing
OPEN(UNIT=1,FILE='ODFluxNodal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=2,FILE='ODElasDisp.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=3,FILE='ODDiffDisp.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=4,FILE='ODTempRise.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=5,FILE='ODDNodePos.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=6,FILE='ODDens.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
!OPEN(UNIT=7,FILE='ODElasDens.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
!OPEN(UNIT=8,FILE='FluxNodalRefl.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
!OPEN(UNIT=9,FILE='KDiffFinal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
!OPEN(UNIT=10,FILE='MDiffFinal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
!OPEN(UNIT=11,FILE='FDiffFinal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
!OPEN(UNIT=12,FILE='ODKElasFinal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
!OPEN(UNIT=13,FILE='ODMElasFinal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
!OPEN(UNIT=14,FILE='ODFElasFinal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=15,FILE='ODDataVector.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=16,FILE='ODPower.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)

! start main program block while loop
TimeChunks = 0
farout: DO
        IF ( TimeChunks > NumTimeChunks ) EXIT
        IF ( Temp(1) > 900._DBL ) EXIT
        ! for loop structure to set parameter array values

                inner1: DO q = 1,NElem+1
                        Ncc(q) = Dens(q)*(1._DBL/AtomicWeight(q))*(6.022D+23) ! atoms per
cc vector
                        IF ( q < NElem+1 ) THEN
                                SigmaFission(q) = Ncc(q)*sigmaF
                                SigmaAbsorption(q) = Ncc(q)*sigmaA
                                SigmaScatter(q) = Ncc(q)*sigmaS
                        ELSE
                                SigmaFission(q) = 0._DBL
                                SigmaAbsorption(q) = Ncc(q)*sigmaAair
                                SigmaScatter(q) = Ncc(q)*sigmaSair
                        END IF

                        DiffHelem(q) = DiffNodePos(q+1) - DiffNodePos(q)

                        DiffusionLength(q) = (1._DBL/3._DBL) / ((SigmaScatter(q))*(1._DBL-
(2._DBL/(3._DBL*AtomicNumber(q)))))
                        ElemCenter(q) = ( DiffNodePos(q) + DiffNodePos(q+1) ) / 2._DBL
                        NeutronSource(q) = SourceTerm/DiffusionLength(q)
                        B1Diff(q) = (1._DBL/DiffusionLength(q)) * (SigmaAbsorption(q) -
FissionNubar*SigmaFission(q))
                        C1Diff(q) = 1._DBL/(NeutronVelocity*DiffusionLength(q))
                        F1Diff(q) = F1DiffCoeff
                END DO inner1

                ! call assembly, BC routines for diffusion solution
                ALLOCATE (KDiff(Ndof+1,Ndof+1), STAT=istat)
                CALL MatrixAssemble2(kdiffelem,DiffNodePos,DiffHelem,B1Diff,KDiff)
                ALLOCATE (MDiff(Ndof+1,Ndof+1), STAT=istat)
                CALL MatrixAssemble2(mdiffelem,DiffNodePos,DiffHelem,C1Diff,MDiff)
                CALL MatrixLump(MDiff,Ndof+1,MDiff)
```

```
                ALLOCATE (FDiff(Ndof+1), STAT=istat)
                CALL VectorAssemble2(fdiffelem,DiffNodePos,DiffHelem,F1Diff,FDiff)
                ALLOCATE (KDiffFinal(Ndof-1,Ndof-1) , STAT=istat)
                ALLOCATE (MDiffFinal(Ndof-1,Ndof-1) , STAT=istat)
                ALLOCATE (FDiffFinal(Ndof-1) , STAT=istat)
                CALL MatrixTruncate1(KDiff,KDiffFinal)
                CALL MatrixTruncate1(MDiff,MDiffFinal)
                CALL VectorTruncate1(FDiff,FDiffFinal)

                DEALLOCATE(KDiff,MDiff,FDiff, STAT=status)
                diffsolverdim = (Ndof-1)
                ALLOCATE (FLoad(diffsolverdim), STAT=istat)
                ALLOCATE (MLoad(diffsolverdim,diffsolverdim), STAT=istat)
                t = 0.
                step = globalstep

                MLoad = MDiffFinal + diffB1*KDiffFinal

                inner2: DO
                        IF ( t > Time ) EXIT
                        FLoad = MATMUL(MDiffFinal,Phiold) -
diffB2*MATMUL(KDiffFinal,Phiold) + (diffB1+diffB2)*FDiffFinal
                        CALL GJSolver(MLoad, FLoad, diffsolverdim, Phinew) ! call solver
here
                        Phiold = Phinew
                        t = t + step
                END DO inner2

                DO i = 1,Ndof+1
                                IF ( i == 1 ) THEN
                                        PhiNodal(i) = 0._DBL
                                ELSE IF ( i == Ndof+1 ) THEN
                                        PhiNodal(i) = 0._DBL
                                ELSE
                                        PhiNodal(i) = Phinew(i-1)
                                END IF
                END DO

                DO i = 1,NElem+1
                        PhiElem(i) = ( PhiNodal(i) + PhiNodal(i+1) ) /
(2._DBL*ElemCenter(i))
                END DO

                DO i = 1,NElem
                        Power(i) = SigmaFission(i)*PhiElem(i)*Volume(i)*EnergyPerFission*EF
                END DO

                DO i = 1,NElem
                        Temp(i) = Temp(i) + (Power(i)*Time) /
(SpecificHeat(i)*Mass(i)*0.001_DBL)
                END DO


                DEALLOCATE (KDiffFinal,FDiffFinal,MDiffFinal,FLoad,MLoad, STAT=status)

                WRITE(1,100) PhiNodal
                100 FORMAT (1X,1000E14.5)

                inner4: DO i = 1,NElem
                        ElasHelem(i) = DiffHelem(i) / 100._DBL
                END DO inner4

                inner41: DO i = 1,NElem
                        IF ( i == 1 ) THEN
                                DTempDR(i) = (Temp(i)-Temp(i+1))/(ElemCenter(i)-
ElemCenter(i+1))
```

```
                        ELSE IF ( i == NElem ) THEN
                                DTempDR(i) = (Temp(i)-Temp(i-1))/(ElemCenter(i)-
ElemCenter(i-1))
                        ELSE
                                DTempDR(i) = (Temp(i+1)-Temp(i-1))/(ElemCenter(i+1)-
ElemCenter(i-1))
                        END IF
                END DO inner41

                inner5: DO i = 1,NElem
                        RefTemp(i) = Temp(i) - 25.
                        YoungsModulus(i) = -1.057457D+8 * Temp(i) + 9.5445D+10
                        LinearExpansion(i) = Temp(i)*9.786468D-9 + 9.918253D-6
                        SpecificHeat(i) = Temp(i)*0.118526_DBL + 104.69832_DBL
                        WaveVelocity(i) = ( (1._DBL-Poisson)*YoungsModulus(i) / &

        ((1._DBL+Poisson)*(1._DBL-2._DBL*Poisson)*(Dens(i)*1000._DBL)) )**(0.5_DBL)
                        C1Elas(i) = 1._DBL / WaveVelocity(i)**2._DBL
                        F1Elas(i) = -
(1._DBL+Poisson)*LinearExpansion(i)*DTempDR(i)*ElasHelem(i)/(1-Poisson)
                        A1Elas(i) = A1ElasCoeff
                END DO inner5

                WRITE(4,200) RefTemp
                200 FORMAT (1X,1000E14.5)
                WRITE(16,1600) Power
                1600 FORMAT (1X,1000E14.5)
                !
                ALLOCATE (KElas(Ndof,Ndof), STAT=istat)
                CALL MatrixAssemble1(kelaselem,ElasNodePos,ElasHelem,A1Elas,KElas)
                ALLOCATE (MElas(Ndof,Ndof), STAT=istat)
                CALL MatrixAssemble1(melaselem,ElasNodePos,ElasHelem,C1Elas,MElas)
                CALL MatrixLump(MElas,Ndof,MElas)
                ALLOCATE (FElas(Ndof), STAT=istat)
                Call VectorAssemble1(felaselem,ElasNodePos,ElasHelem,F1Elas,FElas)

                alphael = ElasNodePos(Ndof)*(1._DBL - Poisson)
                gammael =
((ElasNodePos(Ndof))**2._DBL)*(1._DBL+Poisson)*LinearExpansion(Nelem)*RefTemp(Nelem)
                betael = (3*Poisson - 1._DBL)
                DO r = 1,Ndof
                        IF ( r /= Ndof ) THEN
                                QNeumann(r) = 0._DBL
                        ELSE
                                QNeumann(r) = kel*gammael/alphael
                        END IF
                END DO

                FElas = FElas + QNeumann
                KElas(Ndof,Ndof) = KElas(Ndof,Ndof) + kel*betael/alphael

                ALLOCATE(KElasFinal(Ndof-1,Ndof-1), STAT=istat)
                CALL MatrixTruncate2(KElas,KElasFinal)
                ALLOCATE(MElasFinal(Ndof-1,Ndof-1) , STAT=istat)
                CALL MatrixTruncate2(MElas,MElasFinal)
                ALLOCATE(FElasFinal(Ndof-1), STAT=istat)
                CALL VectorTruncate2(FElas,FElasFinal)
                DEALLOCATE(FElas, STAT=status)
                DEALLOCATE(KElas, STAT=status)
                DEALLOCATE(MElas, STAT=status)

                elassolverdim = Ndof-1
                ALLOCATE(MElasLoad(elassolverdim,elassolverdim), STAT=istat)
                ALLOCATE(FElasLoad(elassolverdim), STAT=istat)
!               KElasHat = KElasFinal + elasa3*MElasFinal
```

```
              ! start while loop structure for elasticity differencing
!             ALLOCATE(TVector(elassolverdim),STAT=istat)

              t = 0.
              step = timestep_elas

!             inner6: DO
!                     IF ( t > Time ) EXIT
!                     Velprev = Vel
!                     Accprev = Acc
!                     Dispprev = Disp
!                     Aprev = AVector
!                     TVector = FElasFinal + MATMUL(MElasFinal,Aprev)
!                     CALL GJSolver(KElasHat, TVector, elassolverdim, Disp)
!                     Acc = elasa3*(Disp-Dispprev) - elasa4*Velprev - elasa5*Accprev
!                     Vel = Velprev + elasa2*Accprev + elasa1*Acc
!                     AVector = elasa3*Disp + elasa4*Vel + elasa5*Acc
!                     t = t + step
!             END DO inner6

              IF ( TimeChunks == 0 ) THEN
                     CALL GJSolver(MElasFinal,FElasFinal,elassolverdim,Accprev)
              END IF

              MElasLoad = MElasFinal + Beta*(step**2._DBL)*KElasFinal

              inner61: DO
                     IF ( t > Time ) EXIT
                            Disppred = Dispprev + step*Velprev +
(step**2._DBL)*(0.5_DBL)*(1._DBL-2._DBL*Beta)*Accprev
                            Velpred = Velprev + (1-Gamma)*step*Accprev
                            FElasLoad = FElasFinal - MATMUL(KElasFinal,Disppred)
                            CALL GJSolver(MElasLoad,FElasLoad,elassolverdim,Accnext)
                            Dispnext = Disppred + Beta*(step**2._DBL)*Accnext
                            Velnext = Velpred + Gamma*step*Accnext
                            Dispprev = Dispnext
                            Velprev = Velnext
                            Accprev = Accnext
                            t = t + step
              END DO inner61

              DEALLOCATE(KElasFinal,STAT=status)
              DEALLOCATE(FElasLoad,MElasLoad,MElasFinal,FElasFinal, STAT=status)

              DO q = 1,Ndof
                     IF ( q == 1 ) THEN
                            ElasDisplacement(q) = 0._DBL
                     ELSE
                            ElasDisplacement(q) = Dispnext(q-1) / ElasNodePos(q)
                     END IF
              END DO

              DO q = 1,Ndof
                     DiffNodePos(q) = 100._DBL*ElasNodePos(q) +
100._DBL*ElasDisplacement(q)
              END DO

              DiffNodePos(Ndof+1) = DiffNodePos(Ndof) + 2.13_DBL*DiffConstantInit

              WRITE(2,300) ElasDisplacement
              300 FORMAT (1X,1000E14.5)
!             WRITE(3,400) DiffDisplacement
!             400 FORMAT (1X,1000E14.5)
              WRITE(5,500) DiffNodePos
              500 FORMAT (1X,1000E14.5)
```

```
               inner9: DO q = 1,NElem+1
                       Volume(q) = (4._DBL*PI/3._DBL) * ( (DiffNodePos(q+1))**3._DBL -
(DiffNodePos(q))**3._DBL )
                       Dens(q) = Mass(q)/Volume(q)
               END DO inner9

               WRITE(6,600) Dens
               600 FORMAT (1X,1000E14.5)
       TimeChunks = TimeChunks + 1
END DO farout

!
! note that the size integers in the matrix outputs below must be
! changed if the mesh size is changed
!

!WRITE(9,900) KDiffFinal
!900 FORMAT (1X, 30E13.3)
!
!WRITE(10,900) MDiffFinal
!
!
!WRITE(12,1000) KElasFinal
!1000 FORMAT (1X, 20E13.3)
!
!WRITE(13,1000) MElasFinal
!
!
!WRITE(11,1100) FDiffFinal
!
!
!WRITE(14,1100) FElasFinal

WRITE(15,1100) Volume
1100 FORMAT (1X, E13.4)



END PROGRAM ODMain
```

Appendix B: Documentation for 1D Code

This appendix contains the documentation for the source code in Appendix A. Explanations of program modules, subroutines, functions and parameters may be found here. The one-dimensional code shall be referred to as *ODMain* in this appendix.

ODMain as seen in Appendix A is a pre-alpha stage code. It is assumed the user has access to the source code and a Fortran 95 compiler. In order to use the code, one merely changes parameters as needed in the global parameter module and recompiles the source code. Output consists of appropriately named data files created in the directory in which your Fortran compiler resides. These output files are overwritten if they already exist in this directory, so be sure to move or copy them elsewhere between successive executions of the code.

B.1 Parameter List and Functionality

The parameters in module ODGlobalConstants control the functionality of *ODMain*. They are listed here with brief descriptions:

- *DBL*: controls the precision of all real variables in the code. Default is 8 for double precision;
- *Nelem*: Number of elements in the elasticity mesh;
- *Ndof*: Number of degrees of freedom in the elasticity mesh;
- *Order*: Order of the shape functions employed plus one. Currently may not be set to anything except 2;
- *PI* : Value of pi. Set to 3.141592653;

- *EF*: Engineering factor to set the fraction of local power that deposits energy as heat. Default value is 1;
- *Radius*: Dimension of the sphere;
- *DensInit*: Initial density for the sphere;
- *DensInitAir*: Initial density for the element outside the sphere;
- *PhiOldInit*: Initial condition for scalar flux;
- *EnergyPerFission*: Energy produced per fission reaction in joules;
- *SourceTerm*: Neutron source term in neutrons per cc per second. Default is zero;
- *Poisson*: Poisson's ratio for the solid material;
- *NeutronVelocity*: One-group average neutron velocity in the system;
- *FissionNubar*: neutrons produced per fission reaction;
- *sigmaF*: microscopic fission cross-section in the solid region;
- *sigmaA*: microscopic absorption cross-section in the solid region;
- *sigmaAair*: microscopic absorption cross-section in the air region;
- *sigmaS*: microscopic scattering cross-section in the solid region;
- *sigmaSair*: microscopic scattering cross-section in the air region;
- *kel* : coefficient of second-order term in elasticity equation in the last element in the solid region;
- *A1DiffCoeff*: coefficient of second-order term in the diffusion equation;
- *F1DiffCoeff*: coefficient of load term in the diffusion equation;
- *A1ElasCoeff*: coefficient of second-order term in the elasticity equation
- *NumTimeChunks*: number of discrete differencing loops solved during program execution;
- *Time*: duration of each differencing loop solved during program execution;
- *globalstep*: step size used within diffusion differencing loop;

- *timestep_elas*: step size used within elasticity differencing loop;
- *Gamma*: parameter used to select elasticity differencing scheme;
- *Beta*: second parameter used to select elasticity differencing scheme;
- *DiffAlpha* : parameter used to select diffusion differencing scheme;
- *DiffConstantInit*: initial diffusion length.

B.2 MODULE LIST

A brief summary of the modules used by the main program block in *ODMain* is as follows:

- **ODGlobalConstants**: The code functionality is controlled from this module by changing parameter values. Every procedure in the code, as well as the main program, uses this module;
- **ODFunctions1**: Contains a pseudo-coordinate transform and the space-dependent coefficient for the zero-order term in the elasticity equation;
- **ODFunctions2**: Contains linear shape functions and their derivatives;
- **ODIntArgs**: Contains subroutines that set the values of the arguments under the integrands in the stiffness, load and mass integrals;
- **ODIntRoutines**: Contains subroutines that implement four-point Newton-Cotes quadrature in one dimension. The subroutines all have the same functionality, but may differ in arguments;
- **ODPreAssembly**: Contains subroutines that bridge the ODIntArgs and ODIntRoutines by actually calling the integration routines on the appropriate arguments;

- **ODAssemblyRoutines**: Contains the subroutines that place the solved integrals in their appropriate locations within stiffness and mass matrices, or load vectors. Also contains some miscellaneous boundary condition routines, and a row-lumping routine for the mass matrix;
- **ODSolverRoutines1**: Contains preliminary solver routines used in ODSolverRoutines2 and ODGaussSolver;
- **ODSolverRoutines2**: Contains final solver routines used in ODGaussSolver;
- **ODGaussSolver**: Implements the Gauss-Jordan solver by calling routines defined in the previous solver routine modules.

B.3 PROCEDURE LIST AND FUNCTIONALITY

Procedures appearing in *ODMain* – functions and subroutines – are listed here in order of appearance. Their arguments, outputs, and functionality are summarized.

Function: Xi

Accepts: spatial coordinate, element number, and element size vector

Functionality: simple function that generates a master element coordinate position solely for shape function generation later in the code.

Returns: Xi

Function: B1ElasCoeff

Accepts: spatial coordinate

Functionality: gives the scalar value of 2 divided by the square of the spatial coordinate

Returns: B1ElasCoeff

Function: Psi

Accepts: element number, shape function number, spatial coordinate, and element size vector

Functionality: Generates linear shape function values

Returns: Psi

Function: dPsidx

Accepts: element number, shape function number, element size vector

Functionality: Generates the derivatives of the linear shape functions

Returns: dPsidx

Subroutine: KelasArg

Accepts: spatial coordinate, element number, shape function number 1, shape function number 2, element size vector, material data vector, dummy name for output scalar

Functionality: Generates the values of the integrand for the elasticity stiffness matrix

Returns: Elasticity stiffness matrix integrands

Subroutine: MelasArg

Accepts: spatial coordinate, element number, shape function number 1, shape function number 2, element size vector, material data vector, dummy name for output scalar

Functionality: Generates the values of the integrand for the elasticity mass matrix

Returns: Elasticity mass matrix integrands

Subroutine: FelasArg

Accepts: spatial coordinate, element number, shape function number, element size vector, material data vector, dummy name for output scalar

Functionality: Generates the values of the integrand for the elasticity load vector

Returns: Elasticity load vector integrands

Subroutine: KdiffArg

Accepts: spatial coordinate, element number, shape function number 1, shape function number 2, element size vector, material data vector, dummy name for output scalar

Functionality: Generates the values of the integrand for the diffusion stiffness matrix

Returns: Diffusion stiffness matrix integrands

Subroutine: MdiffArg

Accepts: spatial coordinate, element number, shape function number 1, shape function number 2, element size vector, material data vector, dummy name for output scalar

Functionality: Generates the values of the integrand for the diffusion mass matrix

Returns: Diffusion mass matrix integrands

Subroutine: FdiffArg

Accepts: spatial coordinate, element number, shape function number, element size vector, material data vector, dummy name for output scalar

Functionality: Generates the values of the integrand for the diffusion load vector

Returns: Diffusion load vector integrands

Subroutine: Integrator1

Accepts: external subroutine generating the values of the function being integrated, vector of nodal positions, vector of element sizes, vector of input data, element number, shape function number 1, shape function number 2, dummy argument for output scalar

Functionality: Calls the subroutine in its arguments, sets a local scalar equal to the function evaluated at a particular point, and integrates the function by summing the contributions from the integration points and multiplying by the appropriate weights. Used for elasticity stiffness and mass integration.

Returns: Scalar value of the integral of the function over the element

Subroutine: Integrator11

Accepts: external subroutine generating the values of the function being integrated, vector of nodal positions, vector of element sizes, vector of input data, element number, shape function number 1, shape function number 2, dummy argument for output scalar

Functionality: The same as Integrator1, except the vector of nodal positions, element sizes and input data are one element larger. Since assumed-shape arrays are used, this requires an extra subroutine. Used for diffusion stiffness and mass integration.

Returns: Scalar value of the integral of the function over the element

Subroutine: Integrator2

Accepts: external subroutine generating the values of the function being integrated, vector of nodal positions, vector of element sizes, vector of input data, element number, shape function number, dummy argument for output scalar

Functionality: Same as Integrator1, except accepts one fewer shape function argument. This integrates the elasticity load vector.

Returns: Scalar value of the integral of the function over the element

Subroutine: Integrator22

Accepts: external subroutine generating the values of the function being integrated, vector of nodal positions, vector of element sizes, vector of input data, element number, shape function number, dummy argument for output scalar

Functionality: Same as Integrator2, except the vectors of nodal positions, element sizes and input data are one element larger. This integrates the diffusion load vector.

Returns: Scalar value of the integral of the function over the element

Subroutine: kdiffelem

Accepts: element number, shape function number 1, shape function number 2, vector of nodal positions, vector of element sizes, vector of input data, dummy argument for output scalar

Functionality: Calls Integrator11 on subroutine KDiffArg.

Returns: Produces contributions to the global diffusion stiffness matrix.

Subroutine: mdiffelem

Accepts: element number, shape function number 1, shape function number 2, vector of nodal positions, vector of element sizes, vector of input data, dummy argument for output scalar

Functionality: Calls Integrator11 on subroutine MDiffArg.

Returns: Produces contributions to the global diffusion mass matrix.

Subroutine: fdiffelem

Accepts: element number, shape function number, vector of nodal positions, vector of element sizes, vector of input data, dummy argument for output scalar

Functionality: Calls Integrator22 on subroutine FDiffArg.

Returns: Produces contributions to the global diffusion load vector.

Subroutine: kelaselem

Accepts: element number, shape function number 1, shape function number 2, vector of nodal positions, vector of element sizes, vector of input data, dummy argument for output scalar

Functionality: Calls Integrator1 on subroutine KElasArg.

Returns: Produces contributions to the global elasticity stiffness matrix.

Subroutine: melaselem

Accepts: element number, shape function number 1, shape function number 2, vector of nodal positions, vector of element sizes, vector of input data, dummy argument for output scalar

Functionality: Calls Integrator1 on subroutine MElasArg.

Returns: Produces contributions to the global elasticity mass matrix.

Subroutine: felaselem

Accepts: element number, shape function number, vector of nodal positions, vector of element sizes, vector of input data, dummy argument for output scalar

Functionality: Calls Integrator2 on subroutine FElasArg.

Returns: Produces contributions to the global elasticity load vector.

Subroutine: MatrixAssemble1

Accepts: external subroutine generating function values to assemble, vector of nodal positions, vector of element sizes, vector of input data, dummy argument for output matrix

Functionality: Loops through elements, loops through shape functions, loops through shape functions again. Calls external subroutine to get the correct contribution to the matrix. Adds this contribution to the appropriate position in the matrix. Only calculates the lower diagonal portion of the matrix; the upper portion is produced via symmetry.

Returns: Produces global mass or stiffness matrix for elasticity

Subroutine: MatrixAssemble2

Accepts: external subroutine generating function values to assemble, vector of nodal positions, vector of element sizes, vector of input data, dummy argument for output matrix

Functionality: The same as MatrixAssemble1, except accepts vectors of nodal positions, element sizes, and input data one element larger.

Returns: Produces global mass or stiffness matrix for diffusion

Subroutine: VectorAssemble1

Accepts: external subroutine generating function values to assemble, vector of nodal positions, vector of element sizes, vector of input data, dummy argument for output vector

Functionality: Loops through elements, loops through shape functions. Calls external subroutine to get the correct contribution to the vector. Adds this contribution to the appropriate position in the vector.

Returns: Produces global load vector for elasticity

Subroutine: VectorAssemble2

Accepts: external subroutine generating function values to assemble, vector of nodal positions, vector of element sizes, vector of input data, dummy argument for output vector

Functionality: Same as VectorAssemble1, except accepts vectors of nodal positions, element sizes, and input data one element larger.

Returns: Produces global load vector for diffusion

Subroutine: MatrixTruncate1

Accepts: Input matrix, dummy argument for output matrix

Functionality: Accepts matrix and truncates the first and last row and column. Used for applying Dirichlet conditions on both domain endpoints.

Returns: Truncated matrix

Subroutine: MatrixTruncate2

Accepts: Input matrix, dummy argument for output matrix

Functionality: Accepts matrix and truncates the first row and column. Used for applying a Dirichlet condition on the first domain endpoint.

Returns: Truncated matrix

Subroutine: VectorTruncate1

Accepts: Input vector, dummy argument for output vector

Functionality: Accepts vector and truncates the first and last entries. Used for applying Dirichlet conditions on both domain endpoints.

Returns: Truncated vector

Subroutine: VectorTruncate2

Accepts: Input vector, dummy argument for output vector

Functionality: Accepts vector and truncates the first entry. Used for applying Dirichlet conditions on the first domain endpoint.

Returns: Truncated vector

Subroutine: MatrixLump

Accepts: Input matrix, matrix dimension, dummy argument for output matrix

Functionality: Adds matrix values in each row together, and sets the diagonal entry for that row equal to the sum. Used to row-lump mass matrices.

Returns: Lumped matrix

Subroutine: Augmented

Accepts: Input matrix, input vector, dimension of square matrix and vector, dummy argument for output matrix

Functionality: Augments the matrix with the input vector.

Returns: Augmented matrix.

Subroutine: RowSwitch

Accepts: Augmented matrix, number of rows in matrix, two rows whose locations in the matrix are to be switched, dummy argument for output matrix

Functionality: Switches the position of two rows in an augmented matrix

Returns: Matrix of the same size and dimension as the input matrix, but with row 1 and row 2 switched.

Subroutine: GetDivFactor

Accepts: Augmented matrix, number of rows in matrix, pivot row number, column number, dummy argument for output vector

Functionality: Accepts the current pivot row and column number in a forward elimination process, and solves for the division factor of each subsequent row after the pivot row.

Returns: Vector of division factors for forward elimination.

Subroutine: SearchForPivot

Accepts: Augmented matrix, number of rows in matrix, row number, column number, dummy argument for output integer

Functionality: Searches the augmented matrix starting at row number and column number for a nonzero entry in that column. It designates the row number of that nonzero entry as the pivot row.

Returns: Pivot row integer for use in forward elimination

Subroutine: ForwardElim

Accepts: Augmented matrix, number of rows in matrix, dummy argument for output matrix

Functionality: Performs a standard forward elimination process on an augmented matrix.

Returns: Upper diagonal matrix ready for backward substitution.

Subroutine: BackSub

Accepts: Forward eliminated matrix, number of rows in matrix, dummy argument for output matrix

Functionality: Performs standard backward substitution on a forward eliminated matrix.

Returns: Augmented matrix in full diagonal form, with solutions to equations occupying the last column.

Subroutine: GJSolver

Accepts: Square matrix of equation coefficients, vector of loads, number of rows in matrix, dummy argument for output vector

Functionality: Calls Augment, ForwardElim, and BackSub in succession.

Returns: The last column of the backward substituted augmented matrix. This is the vector of solutions to the linear system of equations.

Appendix C: Source Code for 2D Code

This is the complete Fortran 95 source code for the two-dimensional code solving the time-dependent neutron diffusion and thermoelasticity equations on a hollow cylindrical domain. It compiles and runs successfully using the g95 Fortran compiler (www.g95.org).

Some text wraps occurred when moved to the word processor software, but these are obvious and can be avoided in reproducing the code.

```
!
!
!
!Software written by Stephen C. Wilson of the University of Texas
!
!This is a finite element code that solves the time-dependent
!coupled neutron diffusion and thermoelasticity equations
!on a hollow cylindrical domain. See dissertation for complete
!documentation.
!
!Developed under contract with Sandia National Laboratories.
!
!June 28, 2006
!
!
!


MODULE TDGlobalConstants
! This module includes data constants
! used throughout
! the main program.

IMPLICIT NONE

! Declaring variables for constant data
INTEGER, PARAMETER :: DBL = 8
REAL(KIND=DBL), PARAMETER :: PI = 3.14159265358979

REAL(KIND=DBL), PARAMETER :: ElasDensSolid = 17040.
REAL(KIND=DBL), PARAMETER :: DiffDensSolid = ElasDensSolid/1000.
REAL(KIND=DBL), PARAMETER :: ZtracSwitch = 0.

REAL(KIND=DBL), PARAMETER :: PhioldInit = 1.D+14
REAL(KIND=DBL), PARAMETER :: TempInit = 25.
REAL(KIND=DBL), PARAMETER :: SourceTerm = 0.
```

```
REAL(KIND=DBL), PARAMETER :: sigmaF = 0.97185D-24

REAL(KIND=DBL), PARAMETER :: sigmaA = 1.1678D-24

REAL(KIND=DBL), PARAMETER :: sigmaAair = 0.06786D-24 * 0.
REAL(KIND=DBL), PARAMETER :: sigmaS = 6.046D-24
REAL(KIND=DBL), PARAMETER :: sigmaSair = 2.438D-24 * 0.000000001
REAL(KIND=DBL), PARAMETER :: NeutronVelocity = 1.D+9

REAL(KIND=DBL), PARAMETER :: FissionNubar = 2.6

REAL(KIND=DBL), PARAMETER :: EF = 1.

REAL(KIND=DBL), PARAMETER :: EnergyPerFission = 320.4D-13
REAL(KIND=DBL), PARAMETER :: Poisson = 0.38

INTEGER, PARAMETER :: NumTimeChunks = 600

REAL(KIND=DBL), PARAMETER :: Time= 5.D-7

REAL(KIND=DBL), PARAMETER :: globalstep = 1.D-7

REAL(KIND=DBL), PARAMETER :: timestep_elas = globalstep
REAL(KIND=DBL), PARAMETER :: alpha = 1./2.

REAL(KIND=DBL), PARAMETER :: Gamma = 1./2.

REAL(KIND=DBL), PARAMETER :: Beta = 1./4.

!REAL(KIND=DBL), PARAMETER :: A1 = alpha * timestep_elas

!REAL(KIND=DBL), PARAMETER :: A2 = (1. - alpha) * timestep_elas
!REAL(KIND=DBL), PARAMETER :: A3 = 1./(Beta * timestep_elas**2.)
!REAL(KIND=DBL), PARAMETER :: A4 = timestep_elas * A3

!REAL(KIND=DBL), PARAMETER :: A5 = (1. / Gamma) - 1.

!REAL(KIND=DBL), PARAMETER :: A6 = alpha / (Beta * timestep_elas)
!REAL(KIND=DBL), PARAMETER :: A7 = (alpha / Beta) - 1.

!REAL(KIND=DBL), PARAMETER :: A8 = ((alpha / Gamma) - 1.) * timestep_elas
REAL(KIND=DBL), PARAMETER :: DiffAlpha = 1.
REAL(KIND=DBL), PARAMETER :: B1 = DiffAlpha*globalstep
REAL(KIND=DBL), PARAMETER :: B2 = (1-DiffAlpha)*globalstep
REAL(KIND=DBL), PARAMETER :: DiffConstantInit = 1.1229
INTEGER, PARAMETER :: DiffNCols = 13
INTEGER, PARAMETER :: DiffNRows = 10
INTEGER, PARAMETER :: DiffNdof = (DiffNCols + 1)*(DiffNRows + 1)
REAL(KIND=DBL), PARAMETER :: ElasHeight = 0.0987 + 0.0005 ! reactivity modified by
changing z-dimension
REAL(KIND=DBL), PARAMETER :: ElasIR = 0.019
REAL(KIND=DBL), PARAMETER :: ElasOR = .1015
REAL(KIND=DBL), PARAMETER :: DiffR = 100._DBL*ElasOR
REAL(KIND=DBL), PARAMETER :: DiffZ = 100._DBL*ElasHeight
INTEGER, PARAMETER :: DiffNelem = DiffNRows*DiffNCols
INTEGER, PARAMETER :: ElasNCols = DiffNCols -4
INTEGER, PARAMETER :: ElasNRows = DiffNRows -1
INTEGER, PARAMETER :: ElasNdof = (ElasNCols +1)*(ElasNRows + 1)*2
INTEGER, PARAMETER :: ZClamped = 0    ! set to (1) to clamp top and bottom, set to (0) to
let top move freely
INTEGER, PARAMETER :: ElasNdofDir = (ElasNCols+1) * (ZClamped + 1) ! elasncols+1
normally, zclamped on doubles it
INTEGER, PARAMETER :: ElasNelem = ElasNRows*ElasNCols
INTEGER, PARAMETER :: DiffNdofReflected = DiffNdof*2 - DiffNCols - 1
INTEGER, PARAMETER :: DiffNdofDir = DiffNRows+DiffNCols+1
INTEGER, PARAMETER :: DiffNdofDirRefl = DiffNRows*2 + DiffNCols*2 + 1
```

```
REAL(KIND=DBL), PARAMETER :: onefourth = 0.25


END MODULE TDGlobalConstants
!
!
!
MODULE TDRowCol
IMPLICIT NONE

!
! functions give element row and column location
! as a function of element number
!
INTEGER, PARAMETER :: testinteger = 100

CONTAINS

FUNCTION DiffRow(elem)
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: elem ! element number
INTEGER :: DiffRow ! row location of element on diffusion mesh
!
REAL :: ratio ! real parameter
INTEGER :: trunc ! truncated parameter
!
ratio = REAL(elem)/REAL(DiffNCols)
trunc = INT(ratio)
!
IF ( trunc - ratio == 0 ) THEN ! checks if elem/DiffNCols is an integer
       DiffRow = elem/DiffNCols     ! if it is, sets diffrow value to the integer
ELSE
       DiffRow = trunc + 1          ! if it isn't, sets diffrow value to the truncation
of ratio plus one
END IF
END FUNCTION DiffRow
!
!
!
FUNCTION DiffCol(elem)
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: elem
INTEGER :: DiffCol ! column location as a function of element number
INTEGER :: DiffRow ! diffrow code is duplicated here to avoid calling it explicitly in
the diffcol function
!
REAL :: ratio ! real parameter
INTEGER :: trunc ! truncated parameter
!
ratio = REAL(elem)/REAL(DiffNCols)
trunc = INT(ratio)
!
IF ( trunc - ratio == 0 ) THEN ! checks if elem/DiffNCols is an integer
       DiffRow = elem/DiffNCols     ! if it is, sets diffrow value to the integer
ELSE
       DiffRow = trunc + 1          ! if it isn't, sets diffrow value to the truncation
of ratio plus one
END IF
IF ( trunc - ratio == 0 ) THEN
       DiffCol = DiffNCols
ELSE
       DiffCol = elem - DiffNCols*(DiffRow - 1)
```

```
END IF
END FUNCTION DiffCol
!
!
!
FUNCTION ElasRow(elem)
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: elem ! element number
INTEGER :: ElasRow ! row location of element on elas mesh
!
REAL :: ratio ! real parameter
INTEGER :: trunc ! truncated parameter
!
ratio = REAL(elem)/REAL(ElasNCols)
trunc = INT(ratio)
!
IF ( trunc - ratio == 0 ) THEN ! checks if elem/ElasNCols is an integer
       ElasRow = elem/ElasNCols     ! if it is, sets row value to the integer
ELSE
       ElasRow = trunc + 1          ! if it isn't, sets row value to the truncation of
ratio plus one
END IF
END FUNCTION ElasRow
!
!
!
FUNCTION ElasCol(elem)
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: elem
INTEGER :: ElasCol ! column location as a function of element number
INTEGER :: ElasRow ! elasrow code is duplicated here to avoid calling it explicitly in
the col function
!
REAL :: ratio ! real parameter
INTEGER :: trunc ! truncated parameter
!
ratio = REAL(elem)/REAL(ElasNCols)
trunc = INT(ratio)
!
IF ( trunc - ratio == 0 ) THEN ! checks if elem/ElasNCols is an integer
       ElasRow = elem/ElasNCols     ! if it is, sets row value to the integer
ELSE
       ElasRow = trunc + 1          ! if it isn't, sets row value to the truncation of
ratio plus one
END IF
IF ( trunc - ratio == 0 ) THEN
       ElasCol = ElasNCols
ELSE
       ElasCol = elem - ElasNCols*(ElasRow - 1)
END IF
END FUNCTION ElasCol
!
!
!
END MODULE TDRowCol
!
!
!
MODULE TDGlobalRoutines
IMPLICIT NONE

CONTAINS
```

```
SUBROUTINE VectorSort(vector,dimVec)
USE TDGlobalConstants
IMPLICIT NONE
!
! sorts Vector integer entries into ascending order,
! output overwrites original vector
! see chapman, pg 266
!
INTEGER, INTENT(IN) :: dimVec
INTEGER, INTENT(INOUT), DIMENSION(dimVec) :: vector
!
INTEGER :: i,j,iswap1 ! indices, temporary swap variable, pointer
INTEGER :: temp
INTEGER, DIMENSION(dimVec) :: iswap
!
outer: DO i = 1,(dimVec-1)
        iswap = MINLOC(vector(i:dimVec))
        iswap1 = iswap(1) + i - 1
        IF ( iswap1 /= i ) THEN
                temp = vector(i)
                vector(i) = vector(iswap1)
                vector(iswap1) = temp
        END IF
!       iptr = i
!               inner: DO j = (i+1),dimVec
!                       minval: IF ( vector(j) < vector(iptr) ) THEN
!                               iptr = j
!                       END IF minval
!               END DO inner
!       swap: IF ( i /= iptr ) THEN
!               temp = vector(i)
!               vector(i) = vector(iptr)
!               vector(iptr) = temp
!       END IF swap
END DO outer
END SUBROUTINE VectorSort

FUNCTION IsSolid(elem)
USE TDGlobalConstants
USE TDRowCol
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: elem ! element being checked
INTEGER :: IsSolid ! yes or no
!
IF ( DiffCol(elem) > 3 .AND. DiffCol(elem) < DiffNCols .AND. DiffRow(elem) < DiffNRows )
THEN
        IsSolid = 1
ELSE
        IsSolid = 0
END IF
END FUNCTION IsSolid

END MODULE TDGlobalRoutines
!
!
!
MODULE TDValueSetRoutines
IMPLICIT NONE
!
!
!
CONTAINS

SUBROUTINE setSolidElements(OutputVector)
```

```
USE TDGlobalConstants
USE TDGlobalRoutines
IMPLICIT NONE
!
INTEGER, INTENT(OUT), DIMENSION(ElasNelem) :: OutputVector
!
INTEGER :: counter,i ! counter, index
!
OutputVector = 0
counter = 1
loop: DO i = 1,DiffNelem
        IF ( IsSolid(i) == 1 ) THEN
                OutputVector(counter) = i
                counter = counter + 1
        ELSE
                CYCLE loop
        END IF
END DO loop
END SUBROUTINE setSolidElements
!
!
!
SUBROUTINE setXNodePosInit(OutputVector,OutputVector2)
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, PARAMETER :: dimOut = DiffNCols + 1
INTEGER, PARAMETER :: dimOut2 = DiffNCols - 3
REAL(KIND=DBL), INTENT(OUT), DIMENSION(DiffNCols + 1) :: OutputVector ! diff nodepos
vector
REAL(KIND=DBL), INTENT(OUT), DIMENSION(DiffNCols - 3) :: OutputVector2 ! elas nodepos
vector
!
INTEGER :: i ! index
!
DO i = 1,dimOut
        OutputVector(i) = 0._DBL
END DO
DO i = 1,dimOut2
        OutputVector2(i) = 0._DBL
END DO
DO i = 1,dimOut
        IF ( i > 0 .AND. i <= 4 ) THEN
                OutputVector(i) = 100._DBL*ElasIR*((REAL(i)-1._DBL)/3._DBL)
        ELSE IF ( i == dimOut ) THEN
                OutputVector(i) = diffR + 2.13_DBL*DiffConstantInit
        ELSE IF ( i == DiffNCols ) THEN
                OutputVector(i) = DiffR
        ELSE
                OutputVector(i) = 100._DBL*ElasIR + (DiffR - 100._DBL*ElasIR)*(REAL(i)-
4._DBL)/(DiffNCols-4._DBL)
        END IF
END DO
DO i = 1,dimOut2
        OutputVector2(i) = (OutputVector(i+3))/100._DBL
END DO
END SUBROUTINE setXNodePosInit
!
!
!
SUBROUTINE setYNodePosInit(OutputVector,OutputVector2)
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, PARAMETER :: dimOut = DiffNRows + 1
INTEGER, PARAMETER :: dimOut2 = DiffNRows
```

```
REAL(KIND=DBL), INTENT(OUT), DIMENSION(DiffNRows + 1) :: OutputVector ! diff nodepos
vector
REAL(KIND=DBL), INTENT(OUT), DIMENSION(DiffNRows) :: OutputVector2 ! elas nodepos vector
!
INTEGER :: i ! index
!
DO i = 1,dimOut
        OutputVector(i) = 0._DBL
END DO
DO i = 1,dimOut2
        OutputVector2(i) = 0._DBL
END DO
DO i = 1,dimOut
        IF ( i == dimOut ) THEN
                OutputVector(i) = diffZ + 2.13_DBL*DiffConstantInit
        ELSE IF ( i == DiffNCols ) THEN
                OutputVector(i) = DiffZ
        ELSE
                OutputVector(i) = DiffZ*(REAL(i)-1._DBL)/(DiffNRows-1._DBL)
        END IF
END DO
DO i = 1,dimOut2
        OutputVector2(i) = (OutputVector(i))/100._DBL
END DO
END SUBROUTINE setYNodePosInit
!
!
!
SUBROUTINE setElasNodePosInit(XVector,YVector,OutputVector)
USE TDGlobalConstants
IMPLICIT NONE
!
REAL(KIND=DBL), INTENT(IN), DIMENSION(DiffNCols-3) :: XVector
REAL(KIND=DBL), INTENT(IN), DIMENSION(DiffNRows) :: YVector
REAL(KIND=DBL), INTENT(OUT), DIMENSION(ElasNdof) :: OutputVector
!
INTEGER :: iter1, iter2, iter3, iter4, i ! while loop counters, index
!
DO i = 1,ElasNdof
        OutputVector(i) = 0
END DO
iter1 = 1
outer: DO
        IF ( iter1 >= (ElasNRows+2) ) EXIT
        iter2 = 1
                inner: DO
                        IF ( iter2 >= (ElasNCols+2) ) EXIT
                        OutputVector((iter2*2-1)+(iter1-1)*(ElasNCols+1)*2) =
XVector(iter2)
                        iter2 = iter2 + 1
                END DO inner
        iter1 = iter1 + 1
END DO outer
iter3 = 1
outer2: DO
        IF ( iter3 >= (ElasNRows+2) ) EXIT
        iter4 = 1
                inner2: DO
                        IF ( iter4 >= (ElasNCols+2) ) EXIT
                        OutputVector((iter4*2)+(iter3-1)*(ElasNCols+1)*2) = YVector(iter3)
                        iter4 = iter4 + 1
                END DO inner2
        iter3 = iter3 + 1
END DO outer2
END SUBROUTINE setElasNodePosInit
!
```

```
!
!
SUBROUTINE setDiffNodePosInit(XVector,YVector,OutputVector)
USE TDGlobalConstants
IMPLICIT NONE
!
REAL(KIND=DBL), INTENT(IN), DIMENSION(DiffNCols+1) :: XVector
REAL(KIND=DBL), INTENT(IN), DIMENSION(DiffNRows-1) :: YVector
REAL(KIND=DBL), INTENT(OUT), DIMENSION(DiffNdof*2) :: OutputVector
!
INTEGER :: iter1, iter2, iter3, iter4, i ! while loop counters, index
!
DO i = 1,(2*DiffNdof)
       OutputVector(i) = 0
END DO
iter1 = 1
outer: DO
       IF ( iter1 >= (DiffNRows+2) ) EXIT
       iter2 = 1
              inner: DO
                     IF ( iter2 >= (DiffNCols+2) ) EXIT
                     OutputVector((iter2*2-1)+(iter1-1)*(DiffNCols+1)*2) =
XVector(iter2)
                     iter2 = iter2 + 1
              END DO inner
       iter1 = iter1 + 1
END DO outer
iter3 = 1
outer2: DO
       IF ( iter3 >= (DiffNRows+2) ) EXIT
       iter4 = 1
              inner2: DO
                     IF ( iter4 >= (DiffNCols+2) ) EXIT
                     OutputVector((iter4*2)+(iter3-1)*(DiffNCols+1)*2) = YVector(iter3)
                     iter4 = iter4 + 1
              END DO inner2
       iter3 = iter3 + 1
END DO outer2
END SUBROUTINE setDiffNodePosInit
!
!
!
SUBROUTINE setDiffDoFDirichlet(OutputVector)
USE TDGlobalConstants
USE TDGlobalRoutines
IMPLICIT NONE
!
INTEGER, PARAMETER :: dimOut = DiffNRows + DiffNCols + 1
INTEGER, INTENT(OUT), DIMENSION(DiffNRows + DiffNCols + 1) :: OutputVector
!
INTEGER, DIMENSION(DiffNCols+1) :: Top
INTEGER, DIMENSION(DiffNRows) :: Right
INTEGER :: i ! index
!
DO i = 1,dimOut
       OutputVector(i) = 0
END DO
DO i = 1,(DiffNCols+1)
       Top(i) = 0
END DO
DO i = 1,DiffNRows
       Right(i) = 0
END DO
DO i = 1,(DiffNCols+1)
       Top(i) = i + DiffNdof - DiffNCols - 1
END DO
```

```
DO i = 1,DiffNRows
        Right(i) = DiffNCols + 1 + (i-1)*(DiffNCols+1)
END DO
DO i = 1,dimOut
        IF ( i <= (DiffNCols+1) ) THEN
                OutputVector(i) = Top(i)
        ELSE IF ( i > (DiffNCols+1) .AND. i <= (DiffNRows+DiffNCols+1) ) THEN
                OutputVector(i) = Right(i-DiffNCols-1)
        ELSE
                OutputVector(i) = 0
        END IF
END DO
CALL VectorSort(OutputVector,dimOut)
END SUBROUTINE setDiffDoFDirichlet
!
!
!
SUBROUTINE setElasDoFDirichlet(OutputVector)
USE TDGlobalConstants
USE TDGlobalRoutines
IMPLICIT NONE
!
INTEGER, INTENT(OUT), DIMENSION(ElasNdofDir) :: OutputVector
!
INTEGER :: i ! index

IF ( ZClamped == 0 ) THEN

        DO i = 1,ElasNdofDir
                OutputVector(i) = 2*i
        END DO

ELSE

        DO i = 1,ElasNdofDir
                IF ( i <= (ElasNCols+1) ) THEN
                        OutputVector(i) = 2*i
                ELSE
                        OutputVector(i) = 2*i + 2*(ElasNCols+1)*(ElasNRows-1)
                END IF
        END DO

END IF

CALL VectorSort(OutputVector,ElasNdofDir)
END SUBROUTINE setElasDoFDirichlet
!
!
!
SUBROUTINE setDiffDoFDirichletReflected(OutputVector)
USE TDGlobalConstants
USE TDGlobalRoutines
IMPLICIT NONE
!
INTEGER, PARAMETER :: dimOut = DiffNdofDirRefl
INTEGER, INTENT(OUT), DIMENSION(DiffNdofDirRefl) :: OutputVector
!
INTEGER, DIMENSION(DiffNCols+1) :: Top,Bottom
INTEGER, DIMENSION(DiffNRows*2-1) :: Right
INTEGER :: i ! index
!
DO i = 1,dimOut
        OutputVector(i) = 0
END DO
DO i = 1,(DiffNCols+1)
        Top(i) = 0
```

```
        Bottom(i) = 0
END DO
DO i = 1,DiffNRows
        Right(i) = 0
END DO
DO i = 1,(DiffNCols+1)
        Top(i) = i + DiffNdofReflected - DiffNCols - 1
        Bottom(i) = i
END DO
DO i = 1,(DiffNRows*2-1)
        Right(i) = DiffNCols + 1 + (i)*(DiffNCols+1)
END DO
DO i = 1,dimOut
        IF ( i <= (DiffNCols+1) ) THEN
                OutputVector(i) = Top(i)
        ELSE IF ( i > (DiffNCols+1) .AND. i <= (DiffNRows*2+DiffNCols) ) THEN
                OutputVector(i) = Right(i-DiffNCols-1)
        ELSE IF ( i > (DiffNRows*2+DiffNCols) ) THEN
                OutputVector(i) = Bottom(i-DiffNCols-2*DiffNRows)
        END IF
END DO
CALL VectorSort(OutputVector,dimOut)
END SUBROUTINE setDiffDoFDirichletReflected
!
!
!
SUBROUTINE setCoeffMat(OutputMatrix)
USE TDGlobalConstants
IMPLICIT NONE
!
REAL(KIND=DBL), INTENT(OUT), DIMENSION(4,4) :: OutputMatrix
!
INTEGER :: i,j ! indices
!
DO i = 1,4
        DO j = 1,4
                OutputMatrix(i,j) = 0._DBL
        END DO
END DO
DO i = 1,3
        DO j = 1,3
                IF ( i == j ) THEN
                OutputMatrix(i,j) = 1._DBL-Poisson
                ELSE
                OutputMatrix(i,j) = Poisson
                END IF
        END DO
END DO
OutputMatrix(4,4) = (1._DBL - 2._DBL*Poisson)/2._DBL
END SUBROUTINE setCoeffMat
!
!
!
END MODULE TDValueSetRoutines
!
!
!
MODULE TDSolverRoutines1
IMPLICIT NONE


CONTAINS
        SUBROUTINE Augmented(InputMatrix,InputVector,ndim,OutputMatrix)
        USE TDGlobalConstants
        IMPLICIT NONE
        INTEGER, INTENT(IN) :: ndim
```

```
REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim) :: InputMatrix
REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim) :: InputVector
REAL(KIND=DBL), INTENT(OUT), DIMENSION(ndim,ndim+1) :: OutputMatrix

INTEGER :: i,j ! indices

OutputMatrix = 0._DBL

DO i = 1,ndim
        DO j = 1,ndim
                OutputMatrix(i,j) = InputMatrix(i,j)
        END DO
END DO

DO i = 1,ndim
        OutputMatrix(i,ndim+1) = InputVector(i)
END DO

END SUBROUTINE Augmented

!
!

SUBROUTINE RowSwitch(InputMatrix,ndim,Row1,Row2,OutputMatrix)
USE TDGlobalConstants
IMPLICIT NONE

INTEGER, INTENT(IN) :: ndim
REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim+1) :: InputMatrix
INTEGER, INTENT(IN) :: Row1,Row2

REAL(KIND=DBL), INTENT(OUT), DIMENSION(ndim,ndim+1) :: OutputMatrix

REAL(KIND=DBL), DIMENSION(ndim,ndim+1) :: tempmatrix
REAL(KIND=DBL), DIMENSION(ndim+1) :: temprow1, temprow2
INTEGER :: i,j,Nrows,Ncols

Nrows = ndim
Ncols = ndim+1

tempmatrix = InputMatrix

DO j = 1,Ncols
        temprow1(j) = InputMatrix(Row1,j)
END DO

DO j = 1,Ncols
        temprow2(j) = InputMatrix(Row2,j)
END DO

DO j = 1,Ncols
        tempmatrix(Row1,j) = temprow2(j)
        tempmatrix(Row2,j) = temprow1(j)
END DO

OutputMatrix = tempmatrix

END SUBROUTINE RowSwitch

!
!

SUBROUTINE GetDivFactor(InputMatrix,ndim,Pivot,ColNum,OutputVector)
USE TDGlobalConstants
IMPLICIT NONE
```

```
        INTEGER, INTENT(IN) :: ndim
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim+1) :: InputMatrix
        INTEGER, INTENT(IN) :: Pivot,ColNum
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(ndim) :: OutputVector

        REAL(KIND=DBL) :: FactorElement
        INTEGER :: i,j

        FactorElement = InputMatrix(Pivot,ColNum)

        DO i = Pivot,ndim
                OutputVector(i) = InputMatrix(i,ColNum) / FactorElement
        END DO

        END SUBROUTINE GetDivFactor

        !
        !

        SUBROUTINE SearchForPivot(InputMatrix,ndim,RowStart,ColNum,OutputInteger)
        USE TDGlobalConstants
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: ndim,RowStart,ColNum
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim+1) :: InputMatrix
        INTEGER, INTENT(OUT) :: OutputInteger

        INTEGER :: pivot,i

        pivot = 1

        DO i = RowStart,ndim
                IF ( InputMatrix(i,ColNum) /= 0. ) THEN
                        pivot = i
                        EXIT
                END IF
        END DO

        OutputInteger = pivot

        END SUBROUTINE SearchForPivot


END MODULE TDSolverRoutines1

MODULE TDSolverRoutines2

IMPLICIT NONE

CONTAINS

        SUBROUTINE ForwardElim(InputMatrix,ndim,OutputMatrix)
        USE TDGlobalConstants
        USE TDSolverRoutines1
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: ndim
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim+1) :: InputMatrix
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(ndim,ndim+1) :: OutputMatrix

        INTEGER :: i,j,Nrows,Ncols,rowstart,PivotRow,istat,status
        REAL(KIND=DBL), DIMENSION(ndim,ndim+1) :: tempmatrixB,tempmatrixQ
        REAL(KIND=DBL) :: FactorElement
        REAL(KIND=DBL), DIMENSION(ndim) :: DivFactor

        Nrows = ndim
```

```
        Ncols = ndim+1
        rowstart = 1

        PivotRow = 1

        tempmatrixB = InputMatrix
        tempmatrixQ = InputMatrix

        DO
                IF ( rowstart > Nrows ) EXIT
                tempmatrixQ = tempmatrixB
                CALL SearchForPivot(tempmatrixB,ndim,rowstart,rowstart,PivotRow)
                IF ( PivotRow /= rowstart ) THEN
                        CALL RowSwitch(tempmatrixB,ndim,PivotRow,rowstart,tempmatrixB)
                        CALL RowSwitch(tempmatrixQ,ndim,PivotRow,rowstart,tempmatrixQ)
                        PivotRow = rowstart
                END IF
                FactorElement = tempmatrixB(PivotRow,rowstart)
                CALL GetDivFactor(tempmatrixB,ndim,PivotRow,rowstart,DivFactor)
                DO i = rowstart,Nrows
                        DO j = rowstart,Ncols
                                IF ( i /= PivotRow ) THEN
                                        tempmatrixB(i,j) = tempmatrixQ(i,j) -
DivFactor(i)*tempmatrixQ(PivotRow,j)
                                ELSE
                                        tempmatrixB(i,j) = tempmatrixQ(i,j) / FactorElement
                                END IF
                        END DO
                END DO
                rowstart = rowstart + 1
        END DO

        OutputMatrix = tempmatrixB

        END SUBROUTINE ForwardElim

        !
        !

        SUBROUTINE BackSub(InputMatrix,ndim,OutputMatrix)
        USE TDGlobalConstants
        USE TDSolverRoutines1
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: ndim
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim+1) :: InputMatrix
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(ndim,ndim+1) :: OutputMatrix

        INTEGER :: i,j,Nrows,Ncols,rowstart,colstart,istat
        REAL(KIND=DBL), DIMENSION(ndim,ndim+1) :: tempmatrixB
        REAL(KIND=DBL) :: DivFact

        Nrows = ndim
        Ncols = ndim+1

        tempmatrixB = InputMatrix

        rowstart = Nrows

        DO
                IF ( rowstart <= 1 ) EXIT
                        DO i = 1,(rowstart-1)
                                DivFact = tempmatrixB(i,rowstart) /
tempmatrixB(rowstart,rowstart)
                                DO j = 1,Ncols
```

```
                                        tempmatrixB(i,j) = tempmatrixB(i,j) -
tempmatrixB(rowstart,j)*DivFact
                                END DO
                        END DO
                rowstart = rowstart - 1
        END DO

        OutputMatrix = tempmatrixB

        END SUBROUTINE BackSub

END MODULE TDSolverRoutines2

!
!
!

MODULE TDGaussSolver
IMPLICIT NONE

CONTAINS

        SUBROUTINE GJSolver(InputMatrix,InputVector,ndim,OutputVector)
        USE TDGlobalConstants
        USE TDSolverRoutines1
        USE TDSolverRoutines2
        IMPLICIT NONE

        INTEGER, INTENT(IN) :: ndim
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim,ndim) :: InputMatrix
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ndim) :: InputVector
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(ndim) :: OutputVector

        REAL(KIND=DBL), DIMENSION(ndim,ndim+1) :: tempmatrix

        INTEGER :: i

        CALL Augmented(InputMatrix,InputVector,ndim,tempmatrix)
        CALL ForwardElim(tempmatrix,ndim,tempmatrix)
        CALL BackSub(tempmatrix,ndim,tempmatrix)

        DO i = 1, ndim
                OutputVector(i) = tempmatrix(i,ndim+1)
        END DO

        END SUBROUTINE GJSolver

END MODULE TDGaussSolver
!
!
!
MODULE TDGlobalFunctions
IMPLICIT NONE
!
! contains psi, dpsi, and indexing functions
!
!
CONTAINS

FUNCTION Psi(jay,Xi,Eta)
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: jay ! function selector
REAL(KIND=DBL), INTENT(IN) :: Xi, Eta ! coordinates on master square
REAL(KIND=DBL) :: Psi ! shape function
```

```
!
IF ( jay == 1 ) THEN
        Psi = onefourth*(1._DBL-Xi)*(1._DBL-Eta)
ELSE IF ( jay == 2 ) THEN
        Psi = onefourth*(1._DBL+Xi)*(1._DBL-Eta)
ELSE IF ( jay == 3 ) THEN
        Psi = onefourth*(1._DBL+Xi)*(1._DBL+Eta)
ELSE IF ( jay == 4 ) THEN
        Psi = onefourth*(1._DBL-Xi)*(1._DBL+Eta)
ELSE
        Psi = 0._DBL
END IF
END FUNCTION Psi
!
!
!
FUNCTION dPsidEta(jay,Xi)
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: jay ! shape function selector
REAL(KIND=DBL), INTENT(IN) :: Xi ! coordinate on master square
REAL(KIND=DBL) :: dPsidEta ! derivative of shape function
!
IF ( jay == 1 ) THEN
        dPsidEta = onefourth*Xi - onefourth
ELSE IF ( jay == 2 ) THEN
        dPsidEta = -onefourth*Xi - onefourth
ELSE IF ( jay == 3 ) THEN
        dPsidEta = onefourth*Xi + onefourth
ELSE IF ( jay == 4 ) THEN
        dPsidEta = -onefourth*Xi + onefourth
ELSE
        dPsidEta = 0.0
END IF
END FUNCTION dPsidEta
!
!
!
FUNCTION dPsidXi(jay,Eta)
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: jay ! shape function selector
REAL(KIND=DBL), INTENT(IN) :: Eta ! coordinate on master square
REAL(KIND=DBL) :: dPsidXi ! derivative of shape function
!
IF ( jay == 1 ) THEN
        dPsidXi = onefourth*Eta - onefourth
ELSE IF ( jay == 2 ) THEN
        dPsidXi = -onefourth*Eta + onefourth
ELSE IF ( jay == 3 ) THEN
        dPsidXi = onefourth*Eta + onefourth
ELSE IF ( jay == 4 ) THEN
        dPsidXi = -onefourth*Eta - onefourth
ELSE
        dPsidXi = 0.0
END IF
END FUNCTION dPsidXi
!
!
!
SUBROUTINE ElasDoF(elem,indexout)
USE TDGlobalConstants
USE TDRowCol
IMPLICIT NONE
```

```
!
INTEGER, INTENT(IN) :: elem
INTEGER, INTENT(OUT), DIMENSION(8) :: indexout
!
INTEGER, DIMENSION(4) :: temp1
INTEGER, DIMENSION(4) :: temp2
INTEGER, DIMENSION(8) :: temp3
INTEGER :: i ! index
!
DO i = 1,4
        temp1(i) = 0
        temp2(i) = 0
END DO
DO i = 1,4
        IF ( i == 1 ) THEN
                temp1(i) = 2*elem + (2*ElasRow(elem) - 3)
        ELSE IF ( i == 2 ) THEN
                temp1(i) = 2*elem + (2*ElasRow(elem) - 3) + 2
        ELSE IF ( i == 3 ) THEN
                temp1(i) = 2*elem + (2*ElasRow(elem) - 3) + 2 + 2*ElasNCols + 2
        ELSE IF ( i == 4 ) THEN
                temp1(i) = 2*elem + (2*ElasRow(elem) - 3) + 2 + 2*ElasNCols
        END IF
END DO
DO i = 1,4
        IF ( i == 1 ) THEN
                temp2(i) = 2*elem + (2*ElasRow(elem) - 3)+ 1
        ELSE IF ( i == 2 ) THEN
                temp2(i) = 2*elem + (2*ElasRow(elem) - 3) + 2 + 1
        ELSE IF ( i == 3 ) THEN
                temp2(i) = 2*elem + (2*ElasRow(elem) - 3) + 2 + 2*ElasNCols + 3
        ELSE IF ( i == 4 ) THEN
                temp2(i) = 2*elem + (2*ElasRow(elem) - 3) + 2 + 2*ElasNCols + 1
        END IF
END DO
DO i = 1,8
        temp3(i) = 0
END DO
DO i = 1,8
        IF ( mod(i,2) == 1 ) THEN
                temp3(i) = temp1((i+1)/2)
        ELSE IF ( mod(i,2) == 0 ) THEN
                temp3(i) = temp2(i/2)
        END IF
END DO
indexout = temp3
END SUBROUTINE ElasDoF
!
!
!
SUBROUTINE DiffDoF(elem,indexout)
USE TDGlobalConstants
USE TDRowCol
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: elem
INTEGER, INTENT(OUT), DIMENSION(4) :: indexout
!
INTEGER, DIMENSION(4) :: temp
INTEGER :: i ! index
!
DO i = 1,4
        temp(i) = 0
END DO
DO i = 1,4
        IF ( i == 1 ) THEN
```

```
                temp(i) = elem + DiffRow(elem) - 1
        ELSE IF ( i == 2 ) THEN
                temp(i) = elem + DiffRow(elem)
        ELSE IF ( i == 3 ) THEN
                temp(i) = elem + DiffRow(elem) + DiffNCols + 1
        ELSE IF ( i == 4 ) THEN
                temp(i) = elem + DiffRow(elem) + DiffNCols
        END IF
END DO
indexout = temp
END SUBROUTINE DiffDoF
!
!
!
SUBROUTINE DiffXIndex(elem,indexout)
USE TDGlobalConstants
USE TDRowCol
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: elem
INTEGER, INTENT(OUT), DIMENSION(4) :: indexout
!
INTEGER, DIMENSION(4) :: temp
INTEGER :: i ! index
!
DO i = 1,4
        temp(i) = 0
END DO
DO i = 1,4
        IF ( i == 1 ) THEN
                temp(i) = (elem + DiffRow(elem) - 1)*2 - 1
        ELSE IF ( i == 2 ) THEN
                temp(i) = (elem + DiffRow(elem))*2 - 1
        ELSE IF ( i == 3 ) THEN
                temp(i) = (elem + DiffRow(elem) + DiffNCols + 1)*2 - 1
        ELSE IF ( i == 4 ) THEN
                temp(i) = (elem + DiffRow(elem) + DiffNCols)*2 - 1
        END IF
END DO
indexout = temp
END SUBROUTINE DiffXIndex
!
!
!
SUBROUTINE DiffYIndex(elem,indexout)
USE TDGlobalConstants
USE TDRowCol
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: elem
INTEGER, INTENT(OUT), DIMENSION(4) :: indexout
!
INTEGER, DIMENSION(4) :: temp
INTEGER :: i ! index
!
DO i = 1,4
        temp(i) = 0
END DO
DO i = 1,4
        IF ( i == 1 ) THEN
                temp(i) = (elem + DiffRow(elem) - 1)*2
        ELSE IF ( i == 2 ) THEN
                temp(i) = (elem + DiffRow(elem))*2
        ELSE IF ( i == 3 ) THEN
                temp(i) = (elem + DiffRow(elem) + DiffNCols + 1)*2
        ELSE IF ( i == 4 ) THEN
```

```
                temp(i) = (elem + DiffRow(elem) + DiffNCols)*2
        END IF
END DO
indexout = temp
END SUBROUTINE DiffYIndex
!
!
!
SUBROUTINE ElasXIndex(elem,indexout)
USE TDGlobalConstants
USE TDRowCol
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: elem
INTEGER, INTENT(OUT), DIMENSION(4) :: indexout
!
INTEGER, DIMENSION(4) :: temp
INTEGER :: i ! index
!
DO i = 1,4
        temp(i) = 0
END DO
DO i = 1,4
        IF ( i == 1 ) THEN
                temp(i) = 2*elem + (2*ElasRow(elem) - 3)
        ELSE IF ( i == 2 ) THEN
                temp(i) = 2*elem + (2*ElasRow(elem) - 3) + 2
        ELSE IF ( i == 3 ) THEN
                temp(i) = 2*elem + (2*ElasRow(elem) - 3) + 2 + 2*ElasNCols + 2
        ELSE IF ( i == 4 ) THEN
                temp(i) = 2*elem + (2*ElasRow(elem) - 3) + 2 + 2*ElasNCols
        END IF
END DO
indexout = temp
END SUBROUTINE ElasXIndex
!
!
!
SUBROUTINE ElasYIndex(elem,indexout)
USE TDGlobalConstants
USE TDRowCol
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: elem
INTEGER, INTENT(OUT), DIMENSION(4) :: indexout
!
INTEGER, DIMENSION(4) :: temp
INTEGER :: i ! index
!
DO i = 1,4
        temp(i) = 0
END DO
DO i = 1,4
        IF ( i == 1 ) THEN
                temp(i) = 2*elem + (2*ElasRow(elem) - 3)+ 1
        ELSE IF ( i == 2 ) THEN
                temp(i) = 2*elem + (2*ElasRow(elem) - 3) + 2 + 1
        ELSE IF ( i == 3 ) THEN
                temp(i) = 2*elem + (2*ElasRow(elem) - 3) + 2 + 2*ElasNCols + 3
        ELSE IF ( i == 4 ) THEN
                temp(i) = 2*elem + (2*ElasRow(elem) - 3) + 2 + 2*ElasNCols + 1
        END IF
END DO
indexout = temp
END SUBROUTINE ElasYIndex
!
```

```
!
!
END MODULE TDGlobalFunctions
!
!
!
MODULE TDTransform1
IMPLICIT NONE
!
! contains the first stage of coordinate transform
!

CONTAINS
!
        FUNCTION X(NPV,dimNPV,Index,elem,Xi,Eta)
        USE TDGlobalConstants
        USE TDGlobalFunctions
        IMPLICIT NONE
        !
        REAL(KIND=DBL) :: X ! x-coordinate
        INTEGER, INTENT(IN) :: dimNPV ! dimension of NP vector
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! node position vector
        INTEGER, INTENT(IN) :: elem ! element number
        REAL(KIND=DBL), INTENT(IN) :: Xi, Eta ! coordinates on master square
        EXTERNAL :: Index ! subroutine passed to generate local index arrays
        !
        INTEGER, DIMENSION(4) :: tempindex
        INTEGER :: i ! index
        !
        X = 0 ! initialize X
        CALL Index(elem,tempindex)
        DO i = 1,4
                X = X + NPV(tempindex(i))*Psi(i,Xi,Eta)
        END DO
        END FUNCTION X
        !
        !
        !
        FUNCTION Y(NPV,dimNPV,Index,elem,Xi,Eta)
        USE TDGlobalConstants
        USE TDGlobalFunctions
        IMPLICIT NONE
        !
        REAL(KIND=DBL) :: Y ! y-coordinate
        INTEGER, INTENT(IN) :: dimNPV ! dimension of NP vector
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! node position vector
        INTEGER, INTENT(IN) :: elem ! element number
        REAL(KIND=DBL), INTENT(IN) :: Xi, Eta ! coordinates on master square
        EXTERNAL :: Index ! subroutine passed to generate local index arrays
        !
        INTEGER, DIMENSION(4) :: tempindex
        INTEGER :: i ! index
        !
        Y = 0 ! initialize Y
        CALL Index(elem,tempindex)
        DO i = 1,4
                Y = Y + NPV(tempindex(i))*Psi(i,Xi,Eta)
        END DO
        END FUNCTION Y
        !
        !
        !
        FUNCTION dXdXi(NPV,dimNPV,Index,elem,Eta)
        USE TDGlobalConstants
        USE TDGlobalFunctions
        IMPLICIT NONE
```

```
!
REAL(KIND=DBL) :: dXdXi ! derivative
INTEGER, INTENT(IN) :: dimNPV ! dimension of NP vector
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! node position vector
INTEGER, INTENT(IN) :: elem ! element number
REAL(KIND=DBL), INTENT(IN) :: Eta ! coordinates on master square
EXTERNAL :: Index ! subroutine passed to generate local index arrays
!
INTEGER, DIMENSION(4) :: tempindex
INTEGER :: i ! index
!
dXdXi = 0 ! initialize dXdXi
CALL Index(elem,tempindex)
DO i = 1,4
       dXdXi = dXdXi + NPV(tempindex(i))*dPsidXi(i,Eta)
END DO
END FUNCTION dXdXi
!
!
!
FUNCTION dYdXi(NPV,dimNPV,Index,elem,Eta)
USE TDGlobalConstants
USE TDGlobalFunctions
IMPLICIT NONE
!
REAL(KIND=DBL) :: dYdXi ! derivative
INTEGER, INTENT(IN) :: dimNPV ! dimension of NP vector
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! node position vector
INTEGER, INTENT(IN) :: elem ! element number
REAL(KIND=DBL), INTENT(IN) :: Eta ! coordinates on master square
EXTERNAL :: Index ! subroutine passed to generate local index arrays
!
INTEGER, DIMENSION(4) :: tempindex
INTEGER :: i ! index
!
dYdXi = 0 ! initialize dYdXi
CALL Index(elem,tempindex)
DO i = 1,4
       dYdXi = dYdXi + NPV(tempindex(i))*dPsidXi(i,Eta)
END DO
END FUNCTION dYdXi
!
!
!
FUNCTION dXdEta(NPV,dimNPV,Index,elem,Xi)
USE TDGlobalConstants
USE TDGlobalFunctions
IMPLICIT NONE
!
REAL(KIND=DBL) :: dXdEta ! derivative
INTEGER, INTENT(IN) :: dimNPV ! dimension of NP vector
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! node position vector
INTEGER, INTENT(IN) :: elem ! element number
REAL(KIND=DBL), INTENT(IN) :: Xi ! coordinates on master square
EXTERNAL :: Index ! subroutine passed to generate local index arrays
!
INTEGER, DIMENSION(4) :: tempindex
INTEGER :: i ! index
!
dXdEta = 0 ! initialize dXdEta
CALL Index(elem,tempindex)
DO i = 1,4
       dXdEta = dXdEta + NPV(tempindex(i))*dPsidEta(i,Xi)
END DO
END FUNCTION dXdEta
!
```

```
        !
        !
        FUNCTION dYdEta(NPV,dimNPV,Index,elem,Xi)
        USE TDGlobalConstants
        USE TDGlobalFunctions
        IMPLICIT NONE
        !
        REAL(KIND=DBL) :: dYdEta ! derivative
        INTEGER, INTENT(IN) :: dimNPV ! dimension of NP vector
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! node position vector
        INTEGER, INTENT(IN) :: elem ! element number
        REAL(KIND=DBL), INTENT(IN) :: Xi ! coordinates on master square
        EXTERNAL :: Index ! subroutine passed to generate local index arrays
        !
        INTEGER, DIMENSION(4) :: tempindex
        INTEGER :: i ! index
        !
        dYdEta = 0 ! initialize dYdEta
        CALL Index(elem,tempindex)
        DO i = 1,4
                dYdEta = dYdEta + NPV(tempindex(i))*dPsidEta(i,Xi)
        END DO
        END FUNCTION dYdEta
        !
        !
        !
END MODULE TDTransform1
!
!
!
MODULE TDJacobian
IMPLICIT NONE

! Contains jacobian and arctransform functions.
! (second stage of coordinate transform)
!

CONTAINS
        FUNCTION Jacobian(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
        USE TDGlobalConstants
        USE TDTransform1
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Xi, Eta
        EXTERNAL :: Index1, Index2 ! calls index-setting routines for local element
        REAL(KIND=DBL) :: Jacobian
        !
        INTEGER :: i,j ! index
        ! INTEGER, DIMENSION(4) :: tempindex1, tempindex2
        ! CALL(Index1,tempindex1)
        ! CALL(Index2,tempindex2)
        ! looks like index routines just pass through here, no need for direct calls
        REAL(KIND=DBL), DIMENSION(2,2) :: JacobianMatrix ! jacobian matrix, determinant is
jacobian
        !
        JacobianMatrix(1,1) = dXdXi(NPV,dimNPV,Index1,elem,Eta)
        JacobianMatrix(1,2) = dXdEta(NPV,dimNPV,Index1,elem,Xi)
        JacobianMatrix(2,1) = dYdXi(NPV,dimNPV,Index2,elem,Eta)
        JacobianMatrix(2,2) = dYdEta(NPV,dimNPV,Index2,elem,Xi)
        Jacobian = JacobianMatrix(1,1)*JacobianMatrix(2,2) -
JacobianMatrix(1,2)*JacobianMatrix(2,1)
        END FUNCTION Jacobian
        !
```

```
!
!
FUNCTION ArcTransformRight(NPV,dimNPV,Index1,Index2,elem)
USE TDGlobalConstants
USE TDTransform1
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: dimNPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
INTEGER, INTENT(IN) :: elem
EXTERNAL :: Index1, Index2 ! calls index-setting routines for local element
REAL(KIND=DBL) :: ArcTransformRight
!
ArcTransformRight = ((dXdEta(NPV,dimNPV,Index1,elem,1._DBL))**2._DBL + &
 (dYdEta(NPV,dimNPV,Index2,elem,1._DBL))**2._DBL)**(0.5_DBL)
END FUNCTION ArcTransformRight
!
!
!
FUNCTION ArcTransformLeft(NPV,dimNPV,Index1,Index2,elem)
USE TDGlobalConstants
USE TDTransform1
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: dimNPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
INTEGER, INTENT(IN) :: elem
EXTERNAL :: Index1, Index2 ! calls index-setting routines for local element
REAL(KIND=DBL) :: ArcTransformLeft
!
ArcTransformLeft = ((dXdEta(NPV,dimNPV,Index1,elem,-1._DBL))**2._DBL + &
 (dYdEta(NPV,dimNPV,Index2,elem,-1._DBL))**2._DBL)**(0.5_DBL)
END FUNCTION ArcTransformLeft
!
!
!
FUNCTION ArcTransformTop(NPV,dimNPV,Index1,Index2,elem)
USE TDGlobalConstants
USE TDTransform1
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: dimNPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
INTEGER, INTENT(IN) :: elem
EXTERNAL :: Index1, Index2 ! calls index-setting routines for local element
REAL(KIND=DBL) :: ArcTransformTop
!
ArcTransformTop = ((dXdXi(NPV,dimNPV,Index1,elem,1._DBL))**2._DBL + &
(dYdXi(NPV,dimNPV,Index2,elem,1._DBL))**2._DBL)**(0.5_DBL)
END FUNCTION ArcTransformTop
!
!
!
FUNCTION ArcTransformBottom(NPV,dimNPV,Index1,Index2,elem)
USE TDGlobalConstants
USE TDTransform1
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: dimNPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
INTEGER, INTENT(IN) :: elem
EXTERNAL :: Index1, Index2 ! calls index-setting routines for local element
REAL(KIND=DBL) :: ArcTransformBottom
!
ArcTransformBottom = ((dXdXi(NPV,dimNPV,Index1,elem,-1._DBL))**2._DBL + &
(dYdXi(NPV,dimNPV,Index2,elem,-1._DBL))**2._DBL)**(0.5_DBL)
```

```
        END FUNCTION ArcTransformBottom
        !
        !
        !
END MODULE TDJacobian
!
!
!
MODULE TDTransform2
IMPLICIT NONE

CONTAINS
        FUNCTION dXidX(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDJacobian
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1, Index2 ! calls subroutines to set local index values
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Xi,Eta
        REAL(KIND=DBL) :: dXidX
        !
        ! local index routines not called directly here, just pass through function
        !
        dXidX =
dYdEta(NPV,dimNPV,Index2,elem,Xi)/Jacobian(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
        END FUNCTION dXidX
        !
        !
        !
        FUNCTION dXidY(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDJacobian
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1, Index2 ! calls subroutines to set local index values
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Xi,Eta
        REAL(KIND=DBL) :: dXidY
        !
        dXidY = -
dXdEta(NPV,dimNPV,Index1,elem,Xi)/Jacobian(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
        END FUNCTION dXidY
        !
        !
        !
        FUNCTION dEtadX(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDJacobian
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1, Index2 ! calls subroutines to set local index values
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Xi,Eta
        REAL(KIND=DBL) :: dEtadX
        !
```

```
        dEtadX = -
dYdXi(NPV,dimNPV,Index2,elem,Eta)/Jacobian(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
        END FUNCTION dEtadX
        !
        !
        !
        FUNCTION dEtadY(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDJacobian
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1, Index2 ! calls subroutines to set local index values
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Xi,Eta
        REAL(KIND=DBL) :: dEtadY
        !
        dEtadY =
dXdXi(NPV,dimNPV,Index1,elem,Eta)/Jacobian(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
        END FUNCTION dEtadY
        !
        !
        !
END MODULE TDTransform2
!
!
!
MODULE TDTransform3
IMPLICIT NONE

CONTAINS

        FUNCTION dPsidX(NPV,dimNPV,Index1,Index2,elem,jay,Xi,Eta)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDTransform2
        USE TDGlobalFunctions
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1, Index2 ! calls subroutines to set local index values
        INTEGER, INTENT(IN) :: elem, jay
        REAL(KIND=DBL), INTENT(IN) :: Xi, Eta
        REAL(KIND=DBL) :: dPsidX
        !
        dPsidX = dPsidXi(jay,Eta)*dXidX(NPV,dimNPV,Index1,Index2,elem,Xi,Eta) &
        + dPsidEta(jay,Xi)*dEtadX(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
        END FUNCTION dPsidX
        !
        !
        !
        FUNCTION dPsidY(NPV,dimNPV,Index1,Index2,elem,jay,Xi,Eta)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDTransform2
        USE TDGlobalFunctions
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1, Index2 ! calls subroutines to set local index values
        INTEGER, INTENT(IN) :: elem, jay
        REAL(KIND=DBL), INTENT(IN) :: Xi, Eta
```

```
        REAL(KIND=DBL) :: dPsidY
        !
        dPsidY = dPsidXi(jay,Eta)*dXidY(NPV,dimNPV,Index1,Index2,elem,Xi,Eta) &
        + dPsidEta(jay,Xi)*dEtadY(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
        END FUNCTION dPsidY
        !
        !
        !
        FUNCTION JacdPsidX(NPV,dimNPV,Index1,Index2,elem,jay,Xi,Eta)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDTransform2
        USE TDGlobalFunctions
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1, Index2 ! calls subroutines to set local index values
        INTEGER, INTENT(IN) :: elem, jay
        REAL(KIND=DBL), INTENT(IN) :: Xi, Eta
        REAL(KIND=DBL) :: JacdPsidX
        !
        JacdPsidX = dPsidXi(jay,Eta)*dYdEta(NPV,dimNPV,Index2,elem,Xi) &
        - dPsidEta(jay,Xi)*dYdXi(NPV,dimNPV,Index2,elem,Eta)
        END FUNCTION JacdPsidX
        !
        !
        !
        FUNCTION JacdPsidY(NPV,dimNPV,Index1,Index2,elem,jay,Xi,Eta)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDTransform2
        USE TDGlobalFunctions
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1, Index2 ! calls subroutines to set local index values
        INTEGER, INTENT(IN) :: elem, jay
        REAL(KIND=DBL), INTENT(IN) :: Xi, Eta
        REAL(KIND=DBL) :: JacdPsidY
        !
        JacdPsidY = -dPsidXi(jay,Eta)*dXdEta(NPV,dimNPV,Index1,elem,Xi) &
        + dPsidEta(jay,Xi)*dXdXi(NPV,dimNPV,Index1,elem,Eta)
        END FUNCTION JacdPsidY
        !
        !
        !
END MODULE TDTransform3
!
!
!
MODULE TDShapeFunctionMatrix
IMPLICIT NONE

CONTAINS

        FUNCTION hVector(Xi,Eta)
        USE TDGlobalConstants
        USE TDGlobalFunctions
        IMPLICIT NONE
        !
        REAL(KIND=DBL), INTENT(IN) :: Xi, Eta
        REAL(KIND=DBL), DIMENSION(2,8) :: hVector
        !
        INTEGER :: i,j ! indices
```

```
!
DO i = 1,2
        DO j = 1,8
                hVector(i,j) = 0._DBL
        END DO
END DO
hVector(1,1) = Psi(1,Xi,Eta)
hVector(2,2) = Psi(1,Xi,Eta)
hVector(1,3) = Psi(2,Xi,Eta)
hVector(2,4) = Psi(2,Xi,Eta)
hVector(1,5) = Psi(3,Xi,Eta)
hVector(2,6) = Psi(3,Xi,Eta)
hVector(1,7) = Psi(4,Xi,Eta)
hVector(2,8) = Psi(4,Xi,Eta)
END FUNCTION hVector
!
!
!
FUNCTION hVectorRight(Eta)
USE TDGlobalConstants
USE TDGlobalFunctions
IMPLICIT NONE
!
REAL(KIND=DBL), INTENT(IN) :: Eta
REAL(KIND=DBL), DIMENSION(2,8) :: hVectorRight
!
INTEGER :: i,j ! indices
!
DO i = 1,2
        DO j = 1,8
                hVectorRight(i,j) = 0._DBL
        END DO
END DO
hVectorRight(1,3) = Psi(2,1._DBL,Eta)
hVectorRight(2,4) = Psi(2,1._DBL,Eta)
hVectorRight(1,5) = Psi(3,1._DBL,Eta)
hVectorRight(2,6) = Psi(3,1._DBL,Eta)
END FUNCTION hVectorRight
!
!
!
FUNCTION hVectorLeft(Eta)
USE TDGlobalConstants
USE TDGlobalFunctions
IMPLICIT NONE
!
REAL(KIND=DBL), INTENT(IN) :: Eta
REAL(KIND=DBL), DIMENSION(2,8) :: hVectorLeft
!
INTEGER :: i,j ! indices
!
DO i = 1,2
        DO j = 1,8
                hVectorLeft(i,j) = 0._DBL
        END DO
END DO
hVectorLeft(1,1) = Psi(1,-1._DBL,Eta)
hVectorLeft(2,2) = Psi(1,-1._DBL,Eta)
hVectorLeft(1,7) = Psi(4,-1._DBL,Eta)
hVectorLeft(2,8) = Psi(4,-1._DBL,Eta)
END FUNCTION hVectorLeft
!
!
!
FUNCTION hVectorTop(Xi)
USE TDGlobalConstants
```

```
USE TDGlobalFunctions
IMPLICIT NONE
!
REAL(KIND=DBL), INTENT(IN) :: Xi
REAL(KIND=DBL), DIMENSION(2,8) :: hVectorTop
!
INTEGER :: i,j ! indices
!
DO i = 1,2
        DO j = 1,8
                hVectorTop(i,j) = 0._DBL
        END DO
END DO
hVectorTop(1,5) = Psi(3,Xi,1._DBL)
hVectorTop(2,6) = Psi(3,Xi,1._DBL)
hVectorTop(1,7) = Psi(4,Xi,1._DBL)
hVectorTop(2,8) = Psi(4,Xi,1._DBL)
END FUNCTION hVectorTop
!
!
!
FUNCTION hVectorBottom(Xi)
USE TDGlobalConstants
USE TDGlobalFunctions
IMPLICIT NONE
!
REAL(KIND=DBL), INTENT(IN) :: Xi
REAL(KIND=DBL), DIMENSION(2,8) :: hVectorBottom
!
INTEGER :: i,j ! indices
!
DO i = 1,2
        DO j = 1,8
                hVectorBottom(i,j) = 0._DBL
        END DO
END DO
hVectorBottom(1,1) = Psi(1,Xi,-1._DBL)
hVectorBottom(2,2) = Psi(1,Xi,-1._DBL)
hVectorBottom(1,3) = Psi(2,Xi,-1._DBL)
hVectorBottom(2,4) = Psi(2,Xi,-1._DBL)
END FUNCTION hVectorBottom
!
!
!
FUNCTION dhMatrix(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
USE TDGlobalConstants
USE TDGlobalFunctions
USE TDTransform1
USE TDTransform3
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: dimNPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
EXTERNAL :: Index1, Index2
INTEGER, INTENT(IN) :: elem
REAL(KIND=DBL), INTENT(IN) :: Xi,Eta
REAL(KIND=DBL), DIMENSION(4,8) :: dhMatrix
!
REAL(KIND=DBL), DIMENSION(4,8) :: tempMatrix1,tempMatrix2
INTEGER :: i,j
!
DO i = 1,4
        DO j = 1,8
                tempMatrix1(i,j) = 0._DBL
                tempMatrix2(i,j) = 0._DBL
        END DO
```

```
        END DO
        DO i = 1,4
                DO j = 1,8
                        IF ( i == 1 .AND. MOD(j,2) == 1 ) THEN
                                tempMatrix1(i,j) =
dPsidX(NPV,dimNPV,Index1,Index2,elem,(j+1)/2,Xi,Eta)
                        ELSE IF ( i == 2 .AND. MOD(j,2) == 1 ) THEN
                                tempMatrix1(i,j) =
(1._DBL/X(NPV,dimNPV,Index1,elem,Xi,Eta))*Psi((j+1)/2,Xi,Eta)
                        ELSE IF ( i == 3 ) THEN
                                tempMatrix1(i,j) = 0
                        ELSE IF ( i == 4 .AND. MOD(j,2) == 1 ) THEN
                                tempMatrix1(i,j) =
dPsidY(NPV,dimNPV,Index1,Index2,elem,(j+1)/2,Xi,Eta)
                        ELSE
                                tempMatrix1(i,j) = tempMatrix1(i,j)
                        END IF
                END DO
        END DO
        DO i = 1,4
                DO j = 1,8
                        IF ( i == 1 ) THEN
                                tempMatrix2(i,j) = 0._DBL
                        ELSE IF ( i == 2 ) THEN
                                tempMatrix2(i,j) = 0._DBL
                        ELSE IF ( i == 3 .AND. MOD(j,2) == 0 ) THEN
                                tempMatrix2(i,j) =
dPsidY(NPV,dimNPV,Index1,Index2,elem,j/2,Xi,Eta)
                        ELSE IF ( i == 4 .AND. MOD(j,2) == 0 ) THEN
                                tempMatrix2(i,j) =
dPsidX(NPV,dimNPV,Index1,Index2,elem,j/2,Xi,Eta)
                        ELSE
                                tempMatrix2(i,j) = tempMatrix2(i,j)
                        END IF
                END DO
        END DO
        dhMatrix = tempMatrix2 + tempMatrix1
        END FUNCTION dhMatrix
        !
        !
        !
END MODULE TDShapeFunctionMatrix
!
!
!
MODULE TDArgMatrix
IMPLICIT NONE

CONTAINS
        SUBROUTINE
ArgumentMatrix(NPV,dimNPV,Index1,Index2,YModulus,elem,Xi,Eta,OutputMatrix)
        USE TDGlobalConstants
        USE TDShapeFunctionMatrix
        USE TDValueSetRoutines
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1,Index2
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Xi, Eta
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: YModulus
        REAL(KIND=DBL), DIMENSION(8,8) :: OutputMatrix
        !
        REAL(KIND=DBL), DIMENSION(4,8) :: tempMatrix1        !dhmatrix
        REAL(KIND=DBL), DIMENSION(8,4) :: tempMatrix2        !dhmatrixtrans
```

```
        REAL(KIND=DBL), DIMENSION(4,4) :: CMatrix         !coefficient matrix
        !
        tempMatrix1 = 0
        tempMatrix2 = 0
        CALL setCoeffMat(CMatrix)
        !
        tempMatrix1 = dhMatrix(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
        tempMatrix2 = TRANSPOSE(tempMatrix1)
        CMatrix = ( YModulus(elem)/( (1._DBL+Poisson)*(1._DBL-2._DBL*Poisson) ) )*CMatrix
        tempMatrix2 = MATMUL(tempMatrix2,CMatrix)
        OutputMatrix = MATMUL(tempMatrix2,tempMatrix1)
        END SUBROUTINE ArgumentMatrix
        !
        !
        !
END MODULE TDArgMatrix
!
!
!
MODULE TDIntArguments
IMPLICIT NONE

CONTAINS

        SUBROUTINE
ElasKIntArgMatrix(NPV,dimNPV,Index1,Index2,YModulus,elem,Xi,Eta,OutputMatrix)
        USE TDGlobalConstants
        USE TDArgMatrix
        USE TDTransform1
        USE TDJacobian
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1,Index2 ! calls subroutines setting local index vectors
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Xi, Eta
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: YModulus
        !
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(8,8) :: OutputMatrix
        !
        CALL ArgumentMatrix(NPV,dimNPV,Index1,Index2,YModulus,elem,Xi,Eta,OutputMatrix)
        OutputMatrix =
OutputMatrix*2._DBL*PI*X(NPV,dimNPV,Index1,elem,Xi,Eta)*Jacobian(NPV,dimNPV,Index1,Index2
,elem,Xi,Eta)
        END SUBROUTINE ElasKIntArgMatrix
        !
        !
        !
        SUBROUTINE
ElasMIntArgMatrix(NPV,dimNPV,Index1,Index2,Density,elem,Xi,Eta,OutputMatrix)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDJacobian
        USE TDShapeFunctionMatrix
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1,Index2 ! calls subroutines setting local index vectors
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Xi, Eta
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: Density
        !
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(8,8) :: OutputMatrix
        REAL(KIND=DBL), DIMENSION(2,8) :: tempMatrix1 ! hVector
```

```
        REAL(KIND=DBL), DIMENSION(8,2) :: tempMatrix2 ! hVectortrans
        !
        tempMatrix1 = hVector(Xi,Eta)
        tempMatrix2 = TRANSPOSE(tempMatrix1)
        !tempMatrix2 = tempMatrix2*Density(elem)
        OutputMatrix = MATMUL(tempMatrix2,tempMatrix1)
        OutputMatrix = OutputMatrix*Density(elem)
        OutputMatrix =
OutputMatrix*2._DBL*PI*X(NPV,dimNPV,Index1,elem,Xi,Eta)*Jacobian(NPV,dimNPV,Index1,Index2
,elem,Xi,Eta)
        END SUBROUTINE ElasMIntArgMatrix
        !
        !
        !
        SUBROUTINE
ElasFIntArgVector(NPV,dimNPV,Index1,Index2,BFVR,BFVZ,elem,Xi,Eta,OutputVector)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDJacobian
        USE TDShapeFunctionMatrix
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1,Index2 ! calls subroutines setting local index vectors
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Xi, Eta
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: BFVR,BFVZ ! body force vector
components
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(8) :: OutputVector
        !
        REAL(KIND=DBL), DIMENSION(8,2) :: tempVector ! hVectortrans
        REAL(KIND=DBL), DIMENSION(2,8) :: tempVector2 ! hVector
        REAL(KIND=DBL), DIMENSION(2) :: tempCoeff  ! body force vector
        !
        tempVector2 = hVector(Xi,Eta)
        tempVector = TRANSPOSE(tempVector2)
        tempCoeff(1) = BFVR(elem)
        tempCoeff(2) = BFVZ(elem)
        !
        OutputVector = MATMUL(tempVector,tempCoeff)
        OutputVector =
OutputVector*2._DBL*PI*X(NPV,dimNPV,Index1,elem,Xi,Eta)*Jacobian(NPV,dimNPV,Index1,Index2
,elem,Xi,Eta)
        END SUBROUTINE ElasFIntArgVector
        !
        !
        !
        SUBROUTINE
ElasTracRightIntArg(NPV,dimNPV,Index1,Index2,TRVR,elem,Eta,OutputVector)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDJacobian
        USE TDShapeFunctionMatrix
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1,Index2 ! calls subroutines setting local index vectors
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Eta
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: TRVR ! body force vector
components
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(8) :: OutputVector
        !
        REAL(KIND=DBL), DIMENSION(8,2) :: tempVector ! hVectorRighttrans
```

```
        REAL(KIND=DBL), DIMENSION(2,8) :: tempVector2 ! hVectorRight
        REAL(KIND=DBL), DIMENSION(2) :: tempCoeff  ! body force vector
        !
        tempVector2 = hVectorRight(Eta)
        tempVector = TRANSPOSE(tempVector2)
        tempCoeff(1) = TRVR(elem)
        tempCoeff(2) = 0
        !
        OutputVector = MATMUL(tempVector,tempCoeff)
        OutputVector =
OutputVector*2._DBL*PI*X(NPV,dimNPV,Index1,elem,1._DBL,Eta)*ArcTransformRight(NPV,dimNPV,
Index1,Index2,elem)
        END SUBROUTINE ElasTracRightIntArg
        !
        !
        !
        SUBROUTINE ElasTracLeftIntArg(NPV,dimNPV,Index1,Index2,TLVR,elem,Eta,OutputVector)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDJacobian
        USE TDShapeFunctionMatrix
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1,Index2 ! calls subroutines setting local index vectors
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Eta
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: TLVR ! body force vector
components
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(8) :: OutputVector
        !
        REAL(KIND=DBL), DIMENSION(8,2) :: tempVector ! hVectorLefttrans
        REAL(KIND=DBL), DIMENSION(2,8) :: tempVector2 ! hVectorLeft
        REAL(KIND=DBL), DIMENSION(2) :: tempCoeff  ! body force vector
        !
        tempVector2 = hVectorLeft(Eta)
        tempVector = TRANSPOSE(tempVector2)
        tempCoeff(1) = TLVR(elem)
        tempCoeff(2) = 0
        !
        OutputVector = MATMUL(tempVector,tempCoeff)
        OutputVector = OutputVector*2._DBL*PI*X(NPV,dimNPV,Index1,elem,-
1._DBL,Eta)*ArcTransformLeft(NPV,dimNPV,Index1,Index2,elem)
        END SUBROUTINE ElasTracLeftIntArg
        !
        !
        !
        SUBROUTINE ElasTracTopIntArg(NPV,dimNPV,Index1,Index2,TTVZ,elem,Xi,OutputVector)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDJacobian
        USE TDShapeFunctionMatrix
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1,Index2 ! calls subroutines setting local index vectors
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Xi
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: TTVZ ! body force vector
components
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(8) :: OutputVector
        !
        REAL(KIND=DBL), DIMENSION(8,2) :: tempVector ! hVectorToptrans
        REAL(KIND=DBL), DIMENSION(2,8) :: tempVector2 ! hVectorTop
```

```
        REAL(KIND=DBL), DIMENSION(2) :: tempCoeff  ! body force vector
        !
        tempVector2 = hVectorTop(Xi)
        tempVector = TRANSPOSE(tempVector2)
        tempCoeff(1) = 0
        tempCoeff(2) = TTVZ(elem)
        !
        OutputVector = MATMUL(tempVector,tempCoeff)
        OutputVector =
OutputVector*2._DBL*PI*X(NPV,dimNPV,Index1,elem,Xi,1._DBL)*ArcTransformTop(NPV,dimNPV,Ind
ex1,Index2,elem)
        END SUBROUTINE ElasTracTopIntArg
        !
        !
        !
        SUBROUTINE
ElasTracBottomIntArg(NPV,dimNPV,Index1,Index2,TBVZ,elem,Xi,OutputVector)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDJacobian
        USE TDShapeFunctionMatrix
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1,Index2 ! calls subroutines setting local index vectors
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Xi
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: TBVZ ! body force vector
components
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(8) :: OutputVector
        !
        REAL(KIND=DBL), DIMENSION(8,2) :: tempVector ! hVectorBottomtrans
        REAL(KIND=DBL), DIMENSION(2,8) :: tempVector2 ! hVectorBottom
        REAL(KIND=DBL), DIMENSION(2) :: tempCoeff  ! body force vector
        !
        tempVector2 = hVectorBottom(Xi)
        tempVector = TRANSPOSE(tempVector2)
        tempCoeff(1) = 0
        tempCoeff(2) = TBVZ(elem)
        !
        OutputVector = MATMUL(tempVector,tempCoeff)
        OutputVector = OutputVector*2._DBL*PI*X(NPV,dimNPV,Index1,elem,Xi,-
1._DBL)*ArcTransformBottom(NPV,dimNPV,Index1,Index2,elem)
        END SUBROUTINE ElasTracBottomIntArg
        !
        !
        !
        SUBROUTINE VolArg(NPV,dimNPV,Index1,Index2,elem,Xi,Eta,OutputScalar)
        USE TDGlobalConstants
        USE TDTransform1
        USE TDJacobian
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1,Index2 ! calls subroutines setting local index vectors
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(IN) :: Xi,Eta
        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar
        !
        OutputScalar =
2._DBL*PI*X(NPV,dimNPV,Index1,elem,Xi,Eta)*Jacobian(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
        END SUBROUTINE VolArg
        !
        !
```

```
        !
        SUBROUTINE
DiffMIntArgMatrix(NPV,dimNPV,Index1,Index2,DiffCoefficient,elem,aye,jay,Xi,Eta,OutputScal
ar)
        USE TDGlobalConstants
        USE TDGlobalFunctions
        USE TDTransform1
        USE TDJacobian
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1,Index2 ! calls subroutines setting local index vectors
        INTEGER, INTENT(IN) :: elem,aye,jay
        REAL(KIND=DBL), INTENT(IN) :: Xi,Eta
        REAL(KIND=DBL), INTENT(IN), DIMENSION(DiffNelem) :: DiffCoefficient
        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar
        !
        OutputScalar = (DiffCoefficient(elem))*Psi(jay,Xi,Eta)*Psi(aye,Xi,Eta) &
        *2._DBL*PI*X(NPV,dimNPV,Index1,elem,Xi,Eta)*Jacobian(NPV,dimNPV,Index1,Index2,elem
,Xi,Eta)

        ! without 2*PI*R, just as a check (is very wrong)

        !OutputScalar = (DiffCoefficient(elem))*Psi(jay,Xi,Eta)*Psi(aye,Xi,Eta) &
        !*Jacobian(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)

        END SUBROUTINE DiffMIntArgMatrix
        !
        !
        !
        SUBROUTINE
DiffKIntArgMatrix(NPV,dimNPV,Index1,Index2,GiantCoefficient,elem,aye,jay,Xi,Eta,OutputSca
lar)
        USE TDGlobalConstants
        USE TDGlobalFunctions
        USE TDTransform1
        USE TDJacobian
        USE TDTransform3
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1,Index2 ! calls subroutines setting local index vectors
        INTEGER, INTENT(IN) :: elem,aye,jay
        REAL(KIND=DBL), INTENT(IN) :: Xi,Eta
        REAL(KIND=DBL), INTENT(IN), DIMENSION(DiffNelem) :: GiantCoefficient
        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar
        !
        !OutputScalar =
2._DBL*PI*X(NPV,dimNPV,Index1,elem,Xi,Eta)*(JacdPsidX(NPV,dimNPV,Index1,Index2,elem,aye,X
i,Eta)* &
        !JacdPsidX(NPV,dimNPV,Index1,Index2,elem,jay,Xi,Eta)+ &
        !JacdPsidY(NPV,dimNPV,Index1,Index2,elem,aye,Xi,Eta)*JacdPsidY(NPV,dimNPV,Index1,I
ndex2,elem,jay,Xi,Eta)+ &
        !Jacobian(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)*GiantCoefficient(elem)*Psi(jay,Xi,
Eta)*Psi(aye,Xi,Eta))
        !
        OutputScalar =
2._DBL*PI*X(NPV,dimNPV,Index1,elem,Xi,Eta)*Jacobian(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
*&
        (
dPsidX(NPV,dimNPV,Index1,Index2,elem,jay,Xi,Eta)*dPsidX(NPV,dimNPV,Index1,Index2,elem,aye
,Xi,Eta)+ &
        dPsidY(NPV,dimNPV,Index1,Index2,elem,aye,Xi,Eta)*dPsidY(NPV,dimNPV,Index1,Index2,e
lem,jay,Xi,Eta)+ &
```

```
        GiantCoefficient(elem)*Psi(jay,Xi,Eta)*Psi(aye,Xi,Eta) )
        END SUBROUTINE DiffKIntArgMatrix
        !
        !
        !
        SUBROUTINE
DiffFIntArgVector(NPV,dimNPV,Index1,Index2,NS,elem,aye,Xi,Eta,OutputScalar)
        USE TDGlobalConstants
        USE TDGlobalFunctions
        USE TDTransform1
        USE TDJacobian
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        EXTERNAL :: Index1,Index2 ! calls subroutines setting local index vectors
        INTEGER, INTENT(IN) :: elem,aye
        REAL(KIND=DBL), INTENT(IN) :: Xi,Eta
        REAL(KIND=DBL), INTENT(IN), DIMENSION(DiffNelem) :: NS ! neutron source
        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar
        !
        OutputScalar =
2._DBL*PI*X(NPV,dimNPV,Index1,elem,Xi,Eta)*Jacobian(NPV,dimNPV,Index1,Index2,elem,Xi,Eta)
*Psi(aye,Xi,Eta)*NS(elem)
        END SUBROUTINE DiffFIntArgVector
        !
        !
        !
END MODULE TDIntArguments
!
!
!
MODULE TDIntegrationRoutines
IMPLICIT NONE

!
! Six separate integration routines are used
! in the main program. They differ in arguments taken. Only
! the boundary integral routine integrates over 1D, the rest
! integrate over the master element square.
!

CONTAINS
        SUBROUTINE
GaussIntegrator(NPV,Index1,Index2,eqncoeff,elem,aye,jay,func,dimNPV,dimeqncoeff,output)
        USE TDGlobalConstants
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV, dimeqncoeff ! array bounds
        INTEGER, INTENT(IN) :: elem, aye, jay ! passed to func
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector
        EXTERNAL :: Index1, Index2 ! calls index subroutines
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimeqncoeff) :: eqncoeff ! material data
passed to func
        EXTERNAL :: func ! subroutine to generate integration function
        REAL(KIND=DBL), INTENT(OUT):: output ! output scalar
        !
        REAL(KIND=DBL) :: intsum ! trash variable for integration sum
        INTEGER :: i,j ! indices
        REAL(KIND=DBL), DIMENSION(2) :: intpoint ! integration points for 2 point gaussian
quad.
        REAL(KIND=DBL) :: tempscalar ! variable to hold output scalar from func
        !
        intpoint(1) = -0.57735027_DBL
        intpoint(2) = -intpoint(1)
        intsum = 0
```

```
        !
        DO i = 1,2
                DO j = 1,2
                        CALL
func(NPV,dimNPV,Index1,Index2,eqncoeff,elem,aye,jay,intpoint(i),intpoint(j),tempscalar)
                        intsum = intsum + tempscalar
                END DO
        END DO
        !
        output = intsum
        END SUBROUTINE GaussIntegrator
        !
        !
        !
        SUBROUTINE VolumeIntegrator(NPV,Index1,Index2,elem,func,dimNPV,output)
        USE TDGlobalConstants
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV ! array bound
        INTEGER, INTENT(IN) :: elem ! passed to func
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector
  EXTERNAL :: Index1, Index2 ! calls index subroutines
  EXTERNAL :: func ! subroutine generates function being integrated
  REAL(KIND=DBL), INTENT(OUT):: output ! output scalar
  !
  REAL(KIND=DBL) :: intsum ! trash variable for integration sum
  INTEGER :: i,j ! indices
  REAL(KIND=DBL), DIMENSION(2) :: intpoint ! integration points for 2 point gaussian
quad.
  REAL(KIND=DBL) :: tempscalar
  !
  intpoint(1) = -0.57735027_DBL
  intpoint(2) = -intpoint(1)
  intsum = 0
  !
  DO i = 1,2
        DO j = 1,2
                CALL
func(NPV,dimNPV,Index1,Index2,elem,intpoint(i),intpoint(j),tempscalar)
                intsum = intsum + tempscalar
        END DO
  END DO
  !
  output = intsum
  END SUBROUTINE VolumeIntegrator
  !
  !
  !
  SUBROUTINE
GaussIntegrator2(NPV,Index1,Index2,eqncoeff,elem,aye,func,dimNPV,dimeqncoeff,output)
  USE TDGlobalConstants
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN) :: dimNPV, dimeqncoeff ! array bounds
  INTEGER, INTENT(IN) :: elem, aye ! passed to func
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector
  EXTERNAL :: Index1, Index2 ! generates index functions
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimeqncoeff) :: eqncoeff ! material data passed
to func
  EXTERNAL :: func ! generates function being integrated
  REAL(KIND=DBL), INTENT(OUT):: output ! output scalar
  !
  REAL(KIND=DBL) :: intsum ! trash variable for integration sum
  INTEGER :: i,j ! indices
  REAL(KIND=DBL), DIMENSION(2) :: intpoint ! integration points for 2 point gaussian
quad.
```

```
  REAL(KIND=DBL) :: tempscalar
  !
  intpoint(1) = -0.57735027_DBL
  intpoint(2) = -intpoint(1)
  intsum = 0
  !
  DO i = 1,2
        DO j = 1,2
                CALL
func(NPV,dimNPV,Index1,Index2,eqncoeff,elem,aye,intpoint(i),intpoint(j),tempscalar)
               intsum = intsum + tempscalar
        END DO
  END DO
  !
  output = intsum
  END SUBROUTINE GaussIntegrator2
  !
  !
  !
  SUBROUTINE
LineIntegrator(NPV,Index1,Index2,eqncoeff,elem,func,dimNPV,dimeqncoeff,output)
  USE TDGlobalConstants
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN) :: dimNPV, dimeqncoeff ! array bounds
  INTEGER, INTENT(IN) :: elem ! passed to func
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector
  EXTERNAL :: Index1, Index2 ! generates index functions
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimeqncoeff) :: eqncoeff ! material data passed
to func
  EXTERNAL :: func ! generates function being integrated, gives 8x1 vector here
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(8) :: output ! output vector
  !
  REAL(KIND=DBL), DIMENSION(8) :: intsum ! trash variable for integration sum
  INTEGER :: i,j ! indices
  REAL(KIND=DBL), DIMENSION(2) :: intpoint ! integration points for 2 point gaussian
quad.
  REAL(KIND=DBL), DIMENSION(8) :: tempvector ! stores func at intpoints
  !
  intpoint(1) = -0.57735027_DBL
  intpoint(2) = -intpoint(1)
  intsum = 0
  !
  DO i = 1,2
        CALL func(NPV,dimNPV,Index1,Index2,eqncoeff,elem,intpoint(i),tempvector)
        intsum = intsum + tempvector
  END DO
  !
  output = intsum
  END SUBROUTINE LineIntegrator
  !
  !
  !
  SUBROUTINE
GaussIntegrator3(NPV,Index1,Index2,eqncoeff,elem,func,dimNPV,dimeqncoeff,output)
  USE TDGlobalConstants
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN) :: dimNPV, dimeqncoeff ! array bounds
  INTEGER, INTENT(IN) :: elem ! passed to func
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector
  EXTERNAL :: Index1, Index2 ! Index functions
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimeqncoeff) :: eqncoeff ! material data passed
to func
  EXTERNAL :: func ! function being integrated, in this case gives an 8x8 matrix
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(8,8) :: output ! output MATRIX
```

```
  !
  REAL(KIND=DBL), DIMENSION(8,8) :: intsum ! trash variable for integration sum
  INTEGER :: i,j ! indices
  REAL(KIND=DBL), DIMENSION(2) :: intpoint ! integration points for 2 point gaussian
quad.
  REAL(KIND=DBL), DIMENSION(8,8) :: tempmatrix ! stores func at intpoints
  !
  intpoint(1) = -0.57735027_DBL
  intpoint(2) = -intpoint(1)
  intsum = 0
  !
  DO i = 1,2
        DO j = 1,2
                CALL
func(NPV,dimNPV,Index1,Index2,eqncoeff,elem,intpoint(i),intpoint(j),tempmatrix)
                intsum = intsum + tempmatrix
        END DO
  END DO
  !
  output = intsum
  END SUBROUTINE GaussIntegrator3
  !
  !
  !
  SUBROUTINE
GaussIntegrator4(NPV,Index1,Index2,eqncoeff1,eqncoeff2,elem,func,dimNPV,dimeqncoeff,outpu
t)
  USE TDGlobalConstants
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN) :: dimNPV, dimeqncoeff! array bounds
  INTEGER, INTENT(IN) :: elem ! passed to func
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector
  EXTERNAL :: Index1, Index2 ! Index functions
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimeqncoeff) :: eqncoeff1, eqncoeff2 ! material
data passed to func
  EXTERNAL :: func ! function being integrated, gives an 8x1 vector
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(8) :: output ! output VECTOR
  !
  REAL(KIND=DBL), DIMENSION(8) :: intsum ! trash variable for integration sum
  INTEGER :: i,j ! indices
  REAL(KIND=DBL), DIMENSION(2) :: intpoint ! integration points for 2 point gaussian
quad.
  REAL(KIND=DBL), DIMENSION(8) :: tempvector
  !
  intpoint(1) = -0.57735027_DBL
  intpoint(2) = -intpoint(1)
  intsum = 0
  !
  DO i = 1,2
        DO j = 1,2
                CALL
func(NPV,dimNPV,Index1,Index2,eqncoeff1,eqncoeff2,elem,intpoint(i),intpoint(j),tempvector
)
                intsum = intsum + tempvector
        END DO
  END DO
  !
  output = intsum
  END SUBROUTINE GaussIntegrator4
  !
  !
  !
END MODULE TDIntegrationRoutines
!
!
```

```
!
MODULE TDPreAssembly
IMPLICIT NONE

CONTAINS
        SUBROUTINE elaskelem(NPV,dimNPV,Modulus,elem,OutputMatrix)
        USE TDIntegrationRoutines
        USE TDIntArguments
        USE TDGlobalFunctions
        USE TDGlobalConstants
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: Modulus
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(8,8) :: OutputMatrix
        !
        CALL
GaussIntegrator3(NPV,ElasXIndex,ElasYIndex,Modulus,elem,ElasKIntArgMatrix,dimNPV,ElasNele
m,OutputMatrix)
        END SUBROUTINE elaskelem
        !
        !
        !
        SUBROUTINE elasmelem(NPV,dimNPV,Density,elem,OutputMatrix)
        USE TDIntegrationRoutines
        USE TDIntArguments
        USE TDGlobalFunctions
        USE TDGlobalConstants
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: Density
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(8,8) :: OutputMatrix
        !
        CALL
GaussIntegrator3(NPV,ElasXIndex,ElasYIndex,Density,elem,ElasMIntArgMatrix,dimNPV,ElasNele
m,OutputMatrix)
        END SUBROUTINE elasmelem
        !
        !
        !
        SUBROUTINE elasfelem(NPV,dimNPV,BFR,BFZ,elem,OutputVector)
        USE TDIntegrationRoutines
        USE TDIntArguments
        USE TDGlobalFunctions
        USE TDGlobalConstants
        IMPLICIT NONE
        !
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: BFR,BFZ
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(8) :: OutputVector
        !
        CALL
GaussIntegrator4(NPV,ElasXIndex,ElasYIndex,BFR,BFZ,elem,ElasFIntArgVector,dimNPV,ElasNele
m,OutputVector)
        END SUBROUTINE elasfelem
        !
        !
        !
        SUBROUTINE elasrightelem(NPV,dimNPV,TracVec,elem,OutputVector)
```

```
USE TDIntegrationRoutines
USE TDIntArguments
USE TDGlobalFunctions
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: dimNPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: TracVec
INTEGER, INTENT(IN) :: elem
REAL(KIND=DBL), INTENT(OUT), DIMENSION(8) :: OutputVector
!
CALL LineIntegrator(NPV,ElasXIndex,ElasYIndex,TracVec,&
elem,ElasTracRightIntArg,dimNPV,ElasNelem,OutputVector)
END SUBROUTINE elasrightelem
!
!
!
SUBROUTINE elasleftelem(NPV,dimNPV,TracVec,elem,OutputVector)
USE TDIntegrationRoutines
USE TDIntArguments
USE TDGlobalFunctions
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: dimNPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: TracVec
INTEGER, INTENT(IN) :: elem
REAL(KIND=DBL), INTENT(OUT), DIMENSION(8) :: OutputVector
!
CALL LineIntegrator(NPV,ElasXIndex,ElasYIndex,TracVec,&
elem,ElasTracLeftIntArg,dimNPV,ElasNelem,OutputVector)
END SUBROUTINE elasleftelem
!
!
!
SUBROUTINE elastopelem(NPV,dimNPV,TracVec,elem,OutputVector)
USE TDIntegrationRoutines
USE TDIntArguments
USE TDGlobalFunctions
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: dimNPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: TracVec
INTEGER, INTENT(IN) :: elem
REAL(KIND=DBL), INTENT(OUT), DIMENSION(8) :: OutputVector
!
CALL LineIntegrator(NPV,ElasXIndex,ElasYIndex,TracVec,&
elem,ElasTracTopIntArg,dimNPV,ElasNelem,OutputVector)
END SUBROUTINE elastopelem
!
!
!
SUBROUTINE elasbottomelem(NPV,dimNPV,TracVec,elem,OutputVector)
USE TDIntegrationRoutines
USE TDIntArguments
USE TDGlobalFunctions
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: dimNPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(ElasNelem) :: TracVec
```

```
INTEGER, INTENT(IN) :: elem
REAL(KIND=DBL), INTENT(OUT), DIMENSION(8) :: OutputVector
!
CALL LineIntegrator(NPV,ElasXIndex,ElasYIndex,TracVec,&
elem,ElasTracBottomIntArg,dimNPV,ElasNelem,OutputVector)
END SUBROUTINE elasbottomelem
!
!
!
SUBROUTINE diffkelem(NPV,dimNPV,GiantCoeff,elem,aye,jay,OutputScalar)
USE TDIntegrationRoutines
USE TDIntArguments
USE TDGlobalFunctions
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: dimNPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(DiffNelem) :: GiantCoeff
INTEGER, INTENT(IN) :: elem,aye,jay
REAL(KIND=DBL), INTENT(OUT) :: OutputScalar
!
CALL GaussIntegrator(NPV,DiffXIndex,DiffYIndex,GiantCoeff,&
elem,aye,jay,DiffKIntArgMatrix,dimNPV,DiffNelem,OutputScalar)
END SUBROUTINE diffkelem
!
!
!
SUBROUTINE diffmelem(NPV,dimNPV,DiffConst,elem,aye,jay,OutputScalar)
USE TDIntegrationRoutines
USE TDIntArguments
USE TDGlobalFunctions
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: dimNPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(DiffNelem) :: DiffConst
INTEGER, INTENT(IN) :: elem,aye,jay
REAL(KIND=DBL), INTENT(OUT) :: OutputScalar
!
CALL GaussIntegrator(NPV,DiffXIndex,DiffYIndex,DiffConst,&
elem,aye,jay,DiffMIntArgMatrix,dimNPV,DiffNelem,OutputScalar)
END SUBROUTINE diffmelem
!
!
!
SUBROUTINE difffelem(NPV,dimNPV,NSource,elem,aye,OutputScalar)
USE TDIntegrationRoutines
USE TDIntArguments
USE TDGlobalFunctions
USE TDGlobalConstants
IMPLICIT NONE
!
INTEGER, INTENT(IN) :: dimNPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
REAL(KIND=DBL), INTENT(IN), DIMENSION(DiffNelem) :: NSource
INTEGER, INTENT(IN) :: elem,aye
REAL(KIND=DBL), INTENT(OUT) :: OutputScalar
!
CALL GaussIntegrator2(NPV,DiffXIndex,DiffYIndex,NSource,&
elem,aye,DiffFIntArgVector,dimNPV,DiffNelem,OutputScalar)
END SUBROUTINE difffelem
!
!
!
```

```
        SUBROUTINE ElemVolume(NPV,dimNPV,Index1,Index2,elem,OutputScalar)
        USE TDIntegrationRoutines
        USE TDIntArguments
        USE TDGlobalFunctions
        USE TDGlobalConstants
        IMPLICIT NONE
        !
        EXTERNAL :: Index1,Index2
        INTEGER, INTENT(IN) :: dimNPV
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV
        INTEGER, INTENT(IN) :: elem
        REAL(KIND=DBL), INTENT(OUT) :: OutputScalar
        !
        CALL VolumeIntegrator(NPV,Index1,Index2,elem,VolArg,dimNPV,OutputScalar)
        END SUBROUTINE ElemVolume

END MODULE TDPreAssembly
!
!
!
MODULE TDAssemblyBC
IMPLICIT NONE

!
! This module contains assorted assembly, boundary condition,
! and miscellaneous matrix and vector routines.
!

CONTAINS
        SUBROUTINE
MatrixAssemble(func,NPV,DoF,Ndof,FuncData,Nelem,dimNPV,Indexbound,OutputMatrix)
        USE TDGlobalConstants
        IMPLICIT NONE
        !
        EXTERNAL :: func ! subroutine for generating stiffness matrix entries
        INTEGER, INTENT(IN) :: dimNPV,Nelem ! array bounds
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Nelem) :: FuncData ! function data vector
        EXTERNAL :: DoF ! generates function giving dof(e)
        INTEGER, INTENT(IN) :: Ndof ! Number of dof
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof,Ndof) :: OutputMatrix ! output
stiffness matrix
        INTEGER, INTENT(IN) :: Indexbound ! Switches subroutine functionality between elas
and diff
        !
        INTEGER, DIMENSION(Indexbound) :: DoFvector ! DoF  vector moved to local variable
        REAL(KIND=DBL), DIMENSION(Ndof,Ndof) :: Q ! local lower-triangular stiffness
matrix
        REAL(KIND=DBL), DIMENSION(Indexbound,Indexbound) :: ElementMatrix ! local element
matrix
        INTEGER :: e,i,j ! indices
        REAL(KIND=DBL) :: tempscalar
        !

        DO i = 1,Ndof
                DO j = 1,Ndof
                        Q(i,j)=0
                END DO
        END DO
        DO e = 1,Nelem
                CALL DoF(e,DoFvector)
                DO i = 1,Indexbound
                        DO j = 1,Indexbound
                                CALL func(NPV,dimNPV,FuncData,e,i,j,tempscalar)
                                ElementMatrix(i,j) = tempscalar ! note func() generates
matrix entries
```

```
                        END DO
                END DO
                DO i = 1,Indexbound
                        DO j = 1,Indexbound
                                IF ( DoFvector(i) >= DoFvector(j) ) THEN
                                        Q(DoFVector(i),DoFVector(j)) =
Q(DoFVector(i),DoFVector(j)) + ElementMatrix(i,j)
                                ELSE
                                        Q(DoFVector(i),DoFVector(j)) = 0
                                END IF
                        END DO
                END DO
        END DO
        DO i = 1,Ndof
                DO j = 1,Ndof
                        IF ( j > i ) THEN
                                OutputMatrix(i,j) = Q(j,i)
                        ELSE
                                OutputMatrix(i,j) = Q(i,j)
                        END IF
                END DO
        END DO
        END SUBROUTINE MatrixAssemble
        !
        !
        !
        SUBROUTINE
StiffnessMatrixAssemble(func,NPV,DoF,Ndof,FuncData,Nelem,dimNPV,Indexbound,OutputMatrix)
        USE TDGlobalConstants
        IMPLICIT NONE
        !
        EXTERNAL :: func ! function for generating entire element stiffness matrices
        INTEGER, INTENT(IN) :: dimNPV,Nelem ! array bounds
        REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector
        REAL(KIND=DBL), INTENT(IN), DIMENSION(Nelem) :: FuncData ! function data vector
        EXTERNAL :: DoF ! generates local DoF vector
        INTEGER, INTENT(IN) :: Ndof ! Number of dof
        REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof,Ndof) :: OutputMatrix ! output
assembled global stiffness matrix
        INTEGER, INTENT(IN) :: Indexbound ! Switches subroutine functionality between elas
and diff
        !
        INTEGER, DIMENSION(Indexbound) :: DoFvector ! DoF function vector moved to local
variable
        REAL(KIND=DBL), DIMENSION(Ndof,Ndof) :: Q ! local lower-triangular stiffness
matrix
        REAL(KIND=DBL), DIMENSION(Indexbound,Indexbound) :: ElementMatrix ! local element
matrix
        INTEGER :: e,i,j ! indices
        !
        DO i = 1,Ndof
                DO j = 1,Ndof
                        Q(i,j)=0
                END DO
        END DO
        DO e = 1,Nelem
                CALL DoF(e,DoFvector)
                CALL func(NPV,dimNPV,FuncData,e,ElementMatrix) ! note func() generates
entire element matrix
                DO i = 1,Indexbound
                        DO j = 1,Indexbound
                                IF ( DoFvector(i) >= DoFvector(j) ) THEN
                                        Q(DoFVector(i),DoFVector(j)) =
Q(DoFVector(i),DoFVector(j)) + ElementMatrix(i,j)
                                ELSE
                                        Q(DoFVector(i),DoFVector(j)) = 0
```

```
                        END IF
                    END DO
                END DO
        END DO
        DO i = 1,Ndof
                DO j = 1,Ndof
                    IF ( j > i ) THEN
                        OutputMatrix(i,j) = Q(j,i)
                    ELSE
                        OutputMatrix(i,j) = Q(i,j)
                    END IF
                END DO
        END DO
        END SUBROUTINE StiffnessMatrixAssemble
        !
        !
        !
        SUBROUTINE
VectorAssemble(func,NPV,DoF,Ndof,BFR,BFZ,Nelem,dimNPV,Indexbound,OutputVector)
        USE TDGlobalConstants
        IMPLICIT NONE
        !
        EXTERNAL :: func ! function for generating entire load vector
        INTEGER, INTENT(IN) :: dimNPV,Nelem ! array bounds
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector
  REAL(KIND=DBL), INTENT(IN), DIMENSION(Nelem) :: BFR ! function data vector
  REAL(KIND=DBL), INTENT(IN), DIMENSION(Nelem) :: BFZ ! function data vector
  EXTERNAL :: DoF !
  INTEGER, INTENT(IN) :: Ndof ! Number of dof
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof) :: OutputVector ! output vector
  INTEGER, INTENT(IN) :: Indexbound ! Switches subroutine functionality between elas and
diff
  !
  INTEGER, DIMENSION(Indexbound) :: DoFvector ! DoF function vector moved to local
variable
  REAL(KIND=DBL), DIMENSION(Ndof) :: Q ! local lower-triangular stiffness vector
  REAL(KIND=DBL), DIMENSION(Indexbound) :: ElementVector ! local element vector
  INTEGER :: e,i ! indices
  !
  DO i = 1,Ndof
        Q(i) = 0.
  END DO
  DO e = 1,Nelem
        CALL func(NPV,dimNPV,BFR,BFZ,e,ElementVector)
        CALL DoF(e,DoFvector)
        DO i = 1,Indexbound
                Q(DoFvector(i)) = Q(DoFvector(i)) + ElementVector(i)
        END DO
  END DO
  OutputVector = Q
  END SUBROUTINE VectorAssemble
        !
        !
        !
        SUBROUTINE
VectorAssemble2(func,NPV,DoF,Ndof,FuncData,Nelem,dimNPV,Indexbound,OutputVector)
        USE TDGlobalConstants
        IMPLICIT NONE
  !
  EXTERNAL :: func ! function for generating vector entries
  INTEGER, INTENT(IN) :: dimNPV, Nelem ! array bounds

  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector

  REAL(KIND=DBL), INTENT(IN), DIMENSION(Nelem) :: FuncData ! function data vector
  EXTERNAL :: DoF !
```

```
  INTEGER, INTENT(IN) :: Ndof ! Number of dof
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof) :: OutputVector ! output vector
  INTEGER, INTENT(IN) :: Indexbound ! Switches subroutine functionality between elas and
diff
  !
  INTEGER, DIMENSION(Indexbound) :: DoFvector ! DoF function vector moved to local
variable
  REAL(KIND=DBL), DIMENSION(Ndof) :: Q ! local vector
  REAL(KIND=DBL), DIMENSION(Indexbound) :: ElementVector ! local element vector
  INTEGER :: e,i,j ! indices
  REAL(KIND=DBL) :: tempscalar ! stores matrix entries for assembly
  !
  DO i = 1,Ndof
        Q(i)=0
  END DO
  DO e = 1,Nelem
        CALL DoF(e,DoFvector)
        DO j = 1,Indexbound
                CALL func(NPV,dimNPV,FuncData,e,j,tempscalar)
                ElementVector(j) = tempscalar
        END DO
        DO i = 1,Indexbound
                Q(DoFvector(i)) = Q(DoFvector(i)) + ElementVector(i)
        END DO
  END DO
  OutputVector = Q
  END SUBROUTINE VectorAssemble2
  !
  !
  !
  SUBROUTINE
BoundaryAssembleRight(func,NPV,DoF,Ndof,TracVec,Nelem,dimNPV,Indexbound,OutputVector)
  USE TDGlobalConstants
  USE TDRowCol
  IMPLICIT NONE
  !
  EXTERNAL :: func ! function for generating vector entries
  INTEGER, INTENT(IN) :: dimNPV, Nelem ! array bounds
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector
  REAL(KIND=DBL), INTENT(IN), DIMENSION(Nelem) :: TracVec ! function data vector
  EXTERNAL :: DoF !
  INTEGER, INTENT(IN) :: Ndof ! Number of dof
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof) :: OutputVector ! output vector
  INTEGER, INTENT(IN) :: Indexbound ! Switches subroutine functionality between elas and
diff
  !
  INTEGER, DIMENSION(Indexbound) :: DoFvector ! DoF function vector moved to local
variable
  REAL(KIND=DBL), DIMENSION(Ndof) :: Q ! local vector
  REAL(KIND=DBL), DIMENSION(Indexbound) :: ElementVector ! local element vector
  INTEGER :: e,i ! indices
  !
  DO i = 1,Ndof
        Q(i)=0
  END DO
  DO e = 1,Nelem
        CALL DoF(e,DoFvector)
        CALL func(NPV,dimNPV,TracVec,e,ElementVector) ! func() is generating the entire
vector here
        DO i = 1,Indexbound
                IF ( ElasCol(e) == ElasNCols ) THEN
                        Q(DoFvector(i)) = Q(DoFvector(i)) + ElementVector(i)
                ELSE
                        Q(DoFvector(i)) = Q(DoFvector(i))
                END IF
        END DO
```

```
  END DO
  OutputVector = Q
  END SUBROUTINE BoundaryAssembleRight
  !
  !
  !
  SUBROUTINE
BoundaryAssembleLeft(func,NPV,DoF,Ndof,TracVec,Nelem,dimNPV,Indexbound,OutputVector)
  USE TDGlobalConstants
  USE TDRowCol
  IMPLICIT NONE
  !
  EXTERNAL :: func ! function for generating vector entries
  INTEGER, INTENT(IN) :: dimNPV, Nelem ! array bounds
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector
  REAL(KIND=DBL), INTENT(IN), DIMENSION(Nelem) :: TracVec ! function data vector
  EXTERNAL :: DoF !
  INTEGER, INTENT(IN) :: Ndof ! Number of dof
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof) :: OutputVector ! output vector
  INTEGER, INTENT(IN) :: Indexbound ! Switches subroutine functionality between elas and
diff
  !
  INTEGER, DIMENSION(Indexbound) :: DoFvector! DoF function vector moved to local
variable
  REAL(KIND=DBL), DIMENSION(Ndof) :: Q ! local vector
  REAL(KIND=DBL), DIMENSION(Indexbound) :: ElementVector ! local element vector
  INTEGER :: e,i ! indices
  !
  DO i = 1,Ndof
       Q(i)=0
  END DO
  DO e = 1,Nelem
       CALL DoF(e,DoFvector)
       CALL func(NPV,dimNPV,TracVec,e,ElementVector) ! func() is generating the entire
vector here
       DO i = 1,Indexbound
              IF ( ElasCol(e) == 1 ) THEN
                     Q(DoFvector(i)) = Q(DoFvector(i)) + ElementVector(i)
              ELSE
                     Q(DoFvector(i)) = Q(DoFvector(i))
              END IF
       END DO
  END DO
  OutputVector = Q
  END SUBROUTINE BoundaryAssembleLeft
  !
  !
  !
  SUBROUTINE
BoundaryAssembleTop(func,NPV,DoF,Ndof,TracVec,Nelem,dimNPV,Indexbound,OutputVector)
  USE TDGlobalConstants
  USE TDRowCol
  IMPLICIT NONE
  !
  EXTERNAL :: func ! function for generating vector entries
  INTEGER, INTENT(IN) :: dimNPV, Nelem ! array bounds
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector
  REAL(KIND=DBL), INTENT(IN), DIMENSION(Nelem) :: TracVec ! function data vector
  EXTERNAL :: DoF !
  INTEGER, INTENT(IN) :: Ndof ! Number of dof
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof) :: OutputVector ! output vector
  INTEGER, INTENT(IN) :: Indexbound ! Switches subroutine functionality between elas and
diff
  !
  INTEGER, DIMENSION(Indexbound) :: DoFvector ! DoF function vector moved to local
variable
```

```
  REAL(KIND=DBL), DIMENSION(Ndof) :: Q ! local vector
  REAL(KIND=DBL), DIMENSION(Indexbound) :: ElementVector ! local element vector
  INTEGER :: e,i ! indices
  !
  DO i = 1,Ndof
        Q(i)=0
  END DO
  DO e = 1,Nelem
        CALL DoF(e,DoFvector)
        CALL func(NPV,dimNPV,TracVec,e,ElementVector) ! func() is generating the entire
vector here
        DO i = 1,Indexbound
                IF ( ElasRow(e) == ElasNRows ) THEN
                        Q(DoFvector(i)) = Q(DoFvector(i)) + ElementVector(i)
                ELSE
                        Q(DoFvector(i)) = Q(DoFvector(i))
                END IF
        END DO
  END DO
  OutputVector = Q
  END SUBROUTINE BoundaryAssembleTop
  !
  !
  !
  SUBROUTINE
BoundaryAssembleBottom(func,NPV,DoF,Ndof,TracVec,Nelem,dimNPV,Indexbound,OutputVector)
  USE TDGlobalConstants
  USE TDRowCol
  IMPLICIT NONE
  !
  EXTERNAL :: func ! function for generating vector entries
  INTEGER, INTENT(IN) :: dimNPV, Nelem ! array bounds
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimNPV) :: NPV ! Node position vector
  REAL(KIND=DBL), INTENT(IN), DIMENSION(Nelem) :: TracVec ! function data vector
  EXTERNAL :: DoF !
  INTEGER, INTENT(IN) :: Ndof ! Number of dof
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof) :: OutputVector ! output vector
  INTEGER, INTENT(IN) :: Indexbound ! Switches subroutine functionality between elas and
diff
  !
  INTEGER, DIMENSION(Indexbound) :: DoFvector ! DoF function vector moved to local
variable
  REAL(KIND=DBL), DIMENSION(Ndof) :: Q ! local vector
  REAL(KIND=DBL), DIMENSION(Indexbound) :: ElementVector ! local element vector
  INTEGER :: e,i ! indices
  !
  DO i = 1,Ndof
        Q(i)=0
  END DO
  DO e = 1,Nelem
        CALL DoF(e,DoFvector)
        CALL func(NPV,dimNPV,TracVec,e,ElementVector) ! func() is generating the entire
vector here
        DO i = 1,Indexbound
                IF ( ElasRow(e) == 1 ) THEN
                        Q(DoFvector(i)) = Q(DoFvector(i)) + ElementVector(i)
                ELSE
                        Q(DoFvector(i)) = Q(DoFvector(i))
                END IF
        END DO
  END DO
  OutputVector = Q
  END SUBROUTINE BoundaryAssembleBottom
  !
  !
  !
```

```
  SUBROUTINE DispFinalize(DispVector,Ndof,DoFDirichlet,NdofDir,OutputVector)
  USE TDGlobalConstants
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN) :: Ndof ! number of dof, dim of output
  INTEGER, INTENT(IN) :: NdofDir ! number of dirichlet dof, dim of DoFDirichlet
  REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof-NdofDir) :: DispVector ! vector of nodal
values
  INTEGER, INTENT(IN), DIMENSION(NdofDir) :: DoFDirichlet ! vector of dirichlet dof
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof) :: OutputVector ! vector of nodal values
with dirichlet values rD-inserted
  !
  !INTEGER, PARAMETER :: dimDisp = Ndof-NdofDir ! dimension of dispvector, can't do this
in initialization
  REAL(KIND=DBL), DIMENSION(Ndof) :: V ! temp vector
  INTEGER :: i,c ! indices
  !
  DO i = 1,Ndof
        V(i) = 0
  END DO
  c = 1
  DO i = 1,Ndof
        IF ( c < (NdofDir+1) ) THEN
                IF ( i == DoFDirichlet(c) ) THEN
                        V(i) = 0
                        c = c + 1
                ELSE
                        V(i) = DispVector(i-c+1)
                END IF
        ELSE
                V(i) = DispVector(i-c+1)
        END IF
  END DO
  OutputVector = V
  END SUBROUTINE DispFinalize
  !
  !
  !
  SUBROUTINE MatrixLump(SquareMatrix,dimMatrix,OutputMatrix)
  USE TDGlobalConstants
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN) :: dimMatrix ! dimension of square matrix to be row-lumped
  REAL(KIND=DBL), INTENT(IN), DIMENSION(dimMatrix,dimMatrix) :: SquareMatrix ! matrix to
be row-lumped
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(dimMatrix,dimMatrix) :: OutputMatrix ! row-
lumped matrix
  !
  REAL(KIND=DBL), DIMENSION(dimMatrix) :: RowSum ! sums row entries, condenses to this
vector
  REAL(KIND=DBL), DIMENSION(dimMatrix,dimMatrix) :: P ! temp matrix
  INTEGER :: i,j ! indices
  !
  DO i = 1,dimMatrix
        RowSum(i) = 0
  END DO
  DO i = 1,dimMatrix
        DO j = 1,dimMatrix
                P(i,j) = 0
        END DO
  END DO
  DO i = 1,dimMatrix
        DO j = 1,dimMatrix
                RowSum(i) = RowSum(i) + SquareMatrix(i,j)
        END DO
  END DO
```

```
  DO i = 1,dimMatrix
        P(i,i) = RowSum(i)
  END DO
  OutputMatrix = P
  END SUBROUTINE MatrixLump
  !
  !
  !
  SUBROUTINE DiffusionMatrixReflect(DiffMatrix,OutputMatrix)
  USE TDGlobalConstants
  IMPLICIT NONE
  !
  REAL(KIND=DBL), INTENT(IN), DIMENSION(DiffNdof,DiffNdof) :: DiffMatrix ! note DiffNdof
is a global variable
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(DiffNdofReflected,DiffNdofReflected) ::
OutputMatrix ! same with DiffNdofReflected
  !
  REAL(KIND=DBL) :: DoFRatio, DoFNCols, DoFNRows, ii
  INTEGER, DIMENSION(DiffNdof) :: Map ! Index gymnastics cubed
  REAL(KIND=DBL), DIMENSION(DiffNdof,DiffNdof) :: Q, QReflected ! Temp vectors
  REAL(KIND=DBL), DIMENSION(DiffNdofReflected,DiffNdofReflected) :: QShifted,
QReflectedShifted ! Temp matrices
  REAL(KIND=DBL), DIMENSION(DiffNdof) :: RowIndicator, Shift, DoFColNum
  INTEGER :: i,j ! indices
  !
  DoFNCols = DiffNCols + 1
  DoFNRows = DiffNRows + 1
  !
  DoFNCols = DiffNCols + 1.
  DoFNRows = DiffNRows + 1.
  DoFRatio = DoFNCols/DoFNRows
  !
  DO i = 1,DiffNdof
        ii = REAL(i)
        DoFColNum(i) = MOD((ii-1.),DoFNCols) + 1.
  END DO
  DO i = 1,DiffNdof
        ii = REAL(i)
        RowIndicator(i) = (ii - DoFColNum(i))/(DoFNRows)
  END DO
  DO i = 1,DiffNdof
        Shift(i) = ( (DoFNRows - 1.) - (2.*RowIndicator(i))/(DoFRatio) )*DoFNCols
        Map(i) = INT(Shift(i)) + i
  END DO
  DO i = 1,DiffNdof
        DO j = 1,DiffNdof
                Q(i,j) = DiffMatrix(i,j)
                QReflected(i,j) = DiffMatrix(Map(i),Map(j))
        END DO
  END DO
  DO i = 1,DiffNdofReflected
        DO j = 1,DIffNdofReflected
                IF ( i >= (DiffNdof-DiffNCols) .AND. j>= (DiffNdof-DiffNCols) ) THEN
                        QShifted(i,j) = Q(i-(DiffNdof-DiffNCols)+1,j-(DiffNdof-
DiffNCols)+1)
                ELSE
                        QShifted(i,j) = 0
                END IF
                IF ( i <= DiffNdof .AND. j <= DiffNdof ) THEN
                        QReflectedShifted(i,j) = QReflected(i,j)
                ELSE
                        QReflectedShifted(i,j) = 0
                END IF
        END DO
  END DO
  DO i = 1,DiffNdofReflected
```

```
      DO j = 1,DiffNdofReflected
             OutputMatrix(i,j) = QShifted(i,j) + QReflectedShifted(i,j)
      END DO
  END DO
  END SUBROUTINE DiffusionMatrixReflect
  !
  !
  !
  SUBROUTINE DiffusionVectorReflect(DiffVector,OutputVector)
  USE TDGlobalConstants
  IMPLICIT NONE
  !
  REAL(KIND=DBL), INTENT(IN), DIMENSION(DiffNdof) :: DiffVector ! note DiffNdof is a
global variable
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(DiffNdofReflected) :: OutputVector ! same with
DiffNdofReflected
  !
  REAL(KIND=DBL) :: DoFRatio, DoFNCols, DoFNRows, ii
  INTEGER, DIMENSION(DiffNdof) :: Map ! Index gymnastics cubed
  REAL(KIND=DBL), DIMENSION(DiffNdof) :: Q, QReflected ! Temp vectors
  REAL(KIND=DBL), DIMENSION(DiffNdofReflected) :: QShifted, QReflectedShifted ! Temp
vectors
  REAL(KIND=DBL), DIMENSION(DiffNdof) :: RowIndicator, Shift, DoFColNum
  INTEGER :: i,j ! indices
  !
  DoFNCols = DiffNCols + 1.
  DoFNRows = DiffNRows + 1.
  DoFRatio = DoFNCols/DoFNRows
  !
  DO i = 1,DiffNdof
      ii = REAL(i)
      DoFColNum(i) = MOD((ii-1.),DoFNCols) + 1.
  END DO
  DO i = 1,DiffNdof
      ii = REAL(i)
      RowIndicator(i) = (ii - DoFColNum(i))/(DoFNRows)
  END DO
  DO i = 1,DiffNdof
      Shift(i) = ( (DoFNRows - 1.) - (2.*RowIndicator(i))/(DoFRatio) )*DoFNCols
      Map(i) = INT(Shift(i)) + i
  END DO
  DO i = 1,DiffNdof
      Q(i) = DiffVector(i)
      QReflected(i) = DiffVector(Map(i))
  END DO
  DO i = 1,DiffNdofReflected
      IF ( i >= (DiffNdof-DiffNCols) ) THEN
             QShifted(i) = Q(i-(DiffNdof-DiffNCols)+1)
      ELSE
             QShifted(i) = 0
      END IF
      IF ( i <= DiffNdof ) THEN
             QReflectedShifted(i) = QReflected(i)
      ELSE
             QReflectedShifted(i) = 0
      END IF
  END DO
  DO i = 1,DiffNdofReflected
      OutputVector(i) = QShifted(i) + QReflectedShifted(i)
  END DO
  END SUBROUTINE DiffusionVectorReflect
  !
  !
  !
  SUBROUTINE FinalizeReflected(Vector,OutputVector)
  USE TDGlobalConstants
```

```
  IMPLICIT NONE
  !
  REAL(KIND=DBL), INTENT(IN), DIMENSION(DiffNdofReflected) :: Vector ! nodal values plus
reflected values
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(DiffNdof) :: OutputVector ! nodal values on
positive z nodes
  !
  INTEGER :: i ! index
  !
  DO i = 1,DiffNdof
        OutputVector(i) = Vector(i+(DiffNdof-DiffNCols)-1)
  END DO
  END SUBROUTINE FinalizeReflected
  !
  !
  !
  SUBROUTINE DirichletApply(Matrix,DoFDirichlet,Ndof,NdofDir,OutputMatrix)
  USE TDGlobalConstants
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN) :: Ndof,NdofDir ! dof and dirichlet dof dimensions
  INTEGER, INTENT(IN), DIMENSION(NdofDir) :: DoFDirichlet ! vector of dirichlet dof
  REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof,Ndof) :: Matrix ! stiffness matrix being
modified
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof,Ndof) :: OutputMatrix
  !
  INTEGER :: s,i,j ! indices
  REAL(KIND=DBL), DIMENSION(Ndof,Ndof) :: tempMatrix
  !
  tempMatrix = Matrix
  DO s = 1,NdofDir
        DO i = 1,Ndof
                DO j = 1,Ndof
                        IF ( i == DoFDirichlet(s) ) THEN
                                tempMatrix(i,j) = 0
                        ELSE IF ( j == DoFDirichlet(s) ) THEN
                                tempMatrix(i,j) = 0
                        ELSE
                                tempMatrix(i,j) = tempMatrix(i,j)
                        END IF
                END DO
        END DO
  END DO
  OutputMatrix = tempMatrix
  END SUBROUTINE DirichletApply
  !
  !
  !
  ! SUBROUTINE DirichletApply2(Vector,DoFDirichlet,Ndof,NdofDir)
  ! USE TDGlobalConstants
  ! IMPLICIT NONE
  ! !
  ! INTEGER, INTENT(IN) :: Ndof,NdofDir ! dof and dirichlet dof dimensions
  ! INTEGER, INTENT(IN), DIMENSION(NdofDir) :: DoFDirichlet ! vector of dirichlet dof
  ! REAL(KIND=DBL), INTENT(INOUT), DIMENSION(Ndof) :: Vector ! vector being modified
  ! !
  ! INTEGER :: s,i ! indices
  ! !
  ! DO s = 1,NdofDir
  !     DO i = 1,Ndof
  !             IF ( i == DoFDirichlet(s) ) THEN
  !                     Vector(i) = 0
  !             ELSE
  !                     Vector(i) = Vector(i)
  !             END IF
  !     END DO
```

```
  ! END DO
  ! END SUBROUTINE DirichletApply2
  !
  !
  !
  SUBROUTINE VectorConditioner(Vector,DoFDirichlet,Ndof,NdofDir,OutputVector)
  USE TDGlobalConstants
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN) :: Ndof,NdofDir ! dof numbers
  INTEGER, INTENT(IN), DIMENSION(NdofDir) :: DoFDirichlet ! vector of dirichlet dof
  REAL(KIND=DBL), INTENT(IN), DIMENSION(Ndof) :: Vector ! vector being conditioned
  REAL(KIND=DBL), INTENT(OUT), DIMENSION(Ndof - NdofDir) :: OutputVector ! output vector
  !
  INTEGER :: dimOut
  INTEGER :: i,s,k ! index, counters
        INTEGER :: dimTemp
  INTEGER, DIMENSION(NdofDir+1) :: DirichletVec ! temp vector
  !
  dimOut = Ndof - NdofDir
  dimTemp = NdofDir+1
  DO i = 1,dimTemp
        IF ( i <= NdofDir ) THEN
                DirichletVec(i) = DoFDirichlet(i)
        ELSE
                DirichletVec(i) = DoFDirichlet(NdofDir)
        END IF
  END DO
  s = 1
  k = 1
  loop1: DO
        IF ( k > (dimOut) ) EXIT
        inner1: IF ( (k+s-1) == DirichletVec(s) ) THEN
                s = s + 1
        ELSE
                s = s
        END IF inner1
        inner2: IF ( (k+s-1) /= DirichletVec(s) ) THEN
        OutputVector(k) = Vector(k+s-1)
        k = k + 1
        ELSE
        CYCLE loop1
        END IF inner2
  END DO loop1
  END SUBROUTINE VectorConditioner
  !
  !
  !
  SUBROUTINE MatrixConditioner(Matrix,Ndof,NdofDir,OutputMatrix)
  USE TDGlobalConstants
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN) :: Ndof, NdofDir
  REAL(KIND=DBL), INTENT(IN), DIMENSION (Ndof,Ndof) :: Matrix ! input stiffness or mass
matrix, post-dirichletapply
  REAL(KIND=DBL), INTENT(OUT), DIMENSION((Ndof - NdofDir),(Ndof - NdofDir)) ::
OutputMatrix ! output matrix, resized
  !
  INTEGER :: i,j,r,c ! indices, counters
  REAL(KIND=DBL), DIMENSION(Ndof) :: RowSum, ColSum ! temp vectors for checking row and
column sums in matrices
  REAL(KIND=DBL), DIMENSION(Ndof,Ndof) :: Q ! temp array for input matrix
  REAL(KIND=DBL), DIMENSION(Ndof-NdofDir,Ndof) :: P
  REAL(KIND=DBL), DIMENSION(Ndof,Ndof-NdofDir) :: T
  REAL(KIND=DBL), DIMENSION(Ndof-NdofDir,Ndof-NdofDir) :: Y
  !
```

```
  Q = Matrix
  P = 0.
  T = 0.
  Y = 0.
  RowSum = 0
  ColSum = 0
        DO i = 1,Ndof
        DO j = 1,Ndof
                RowSum(i) = RowSum(i) + (Q(i,j))**2.
        END DO
  END DO
  r = 1
  loop1: DO i = 1,Ndof
        IF ( RowSum(i) == 0. ) CYCLE loop1
                DO j = 1,Ndof
                        P(r,j) = Q(i,j)
                END DO
                r = r + 1
  END DO loop1
  DO i = 1,Ndof-NdofDir
        DO j = 1,Ndof
                T(j,i) = P(i,j)
        END DO
  END DO
  DO i = 1,Ndof
        DO j = 1,Ndof-NdofDir
                ColSum(i) = ColSum(i) + (T(i,j))**2.
        END DO
  END DO
  c = 1
  loop2: DO i = 1,Ndof
        IF ( ColSum(i) == 0. ) CYCLE loop2
                DO j = 1,Ndof-NdofDir
                        Y(c,j) = T(i,j)
                END DO
                c = c + 1
  END DO loop2
  DO i = 1,Ndof-NdofDir
        DO j = 1,Ndof-NdofDir
                OutputMatrix(i,j) = Y(j,i)
        END DO
  END DO
  END SUBROUTINE MatrixConditioner
!
END MODULE TDAssemblyBC
!
!
!


PROGRAM testmain
!
USE TDGlobalConstants
USE TDRowCol
USE TDGlobalFunctions
USE TDAssemblyBC
USE TDGaussSolver
USE TDSolverRoutines1
USE TDSolverRoutines2
USE TDGlobalRoutines
USE TDIntArguments
USE TDIntegrationRoutines
USE TDJacobian
USE TDShapeFunctionMatrix
USE TDTransform1
USE TDTransform2
```

```
USE TDTransform3
USE TDValueSetRoutines
USE TDPreAssembly
USE TDArgMatrix
!
IMPLICIT NONE

! variables for main program block
REAL(KIND=DBL) :: t,step ! time counter for inner while loops, step size thereof
INTEGER :: i,j,q,r,s,qq ! indices
INTEGER :: TimeChunks ! counter for main program while loops
! arrays set by subroutine
INTEGER, DIMENSION(ElasNelem) :: SolidElements
REAL(KIND=DBL), DIMENSION(DiffNCols+1) :: DiffXNodePosInit
REAL(KIND=DBL), DIMENSION(DiffNCols-3) :: ElasXNodePosInit
REAL(KIND=DBL), DIMENSION(DiffNRows+1) :: DiffYNodePosInit
REAL(KIND=DBL), DIMENSION(DiffNRows) :: ElasYNodePosInit
REAL(KIND=DBL), DIMENSION(ElasNdof) :: ElasNodePosInit
REAL(KIND=DBL), DIMENSION(2*DiffNdof) :: DiffNodePosInit
INTEGER, DIMENSION(DiffNRows+DiffNCols+1) :: DiffDoFDirichlet
INTEGER, DIMENSION(ElasNdofDir) :: ElasDoFDirichlet
! other variables
INTEGER :: diffsolverdim, elassolverdim
REAL(KIND=DBL), DIMENSION((DiffNdof-DiffNdofDir)) :: Phiold ! IC for diffusion
REAL(KIND=DBL), DIMENSION(ElasNdof-(ElasNdofDir)) :: Disp,Vel,Acc,AVector ! vectors used
in elas differencing
REAL(KIND=DBL), DIMENSION(ElasNdof-(ElasNdofDir)) :: Dispprev,Velprev,Accprev,Aprev,
StaticCheck
REAL(KIND=DBL), DIMENSION(ElasNdof-(ElasNdofDir)) ::
Disppred,Velpred,Accpred,Dispnext,Velnext,Accnext
REAL(KIND=DBL), DIMENSION(2*DiffNdof) :: DiffNodePos
REAL(KIND=DBL), DIMENSION(ElasNdof) :: ElasNodePos
REAL(KIND=DBL), DIMENSION(ElasNelem) :: Temp, LinearExpansion, SpecificHeat, DTempDR,
DTempDZ,ElasDens,InitialElasVolume
REAL(KIND=DBL), DIMENSION(ElasNelem) :: ElasMass,ElasVolume,RefTemp,YoungsModulus,Power
REAL(KIND=DBL), DIMENSION(DiffNelem) ::
InitialDiffVolume,DiffVolume,DiffDens,AtomicWeight,AtomicNumber,DiffMass
REAL(KIND=DBL), DIMENSION(DiffNelem) ::
Ncc,SigmaFission,SigmaAbsorption,DiffusionLength,NeutronSource,SigmaScatter
REAL(KIND=DBL), DIMENSION(DiffNelem) :: EnormousCoefficient,DiffusionConstant,
ElemRCenter,ElemZCenter

REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:,:) :: KDiff,MDiff

REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:) :: FDiff

REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:,:) :: KDiffFinal,MDiffFinal, MLoad

REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:) :: FDiffFinal, FLoad
REAL(KIND=DBL), DIMENSION((DiffNdof-DiffNdofDir)) ::
PhiNew,PhiPred,FluxVelPrev,FluxVelNext

REAL(KIND=DBL), DIMENSION(DiffNdof) :: PhiNodal
REAL(KIND=DBL), DIMENSION(DiffNelem) :: PhiElem, dPhidR, dPhidZ
REAL(KIND=DBL), DIMENSION(ElasNelem) ::
BodyForceR,BodyForceZ,TractionRightR,TractionLeftR,TractionTopZ, DiffusionElement
REAL(KIND=DBL), DIMENSION(ElasNelem) :: TractionBottomZ

REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:,:) :: KElas, MElas,
KElasFinal,MElasFinal,KElasHat

REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:) :: FElas,ElasRight,ElasLeft,ElasTop,ElasBottom,
FElasFinal, TempVector

REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:) :: FElasLoad
```

```
REAL(KIND=DBL), ALLOCATABLE, DIMENSION(:,:) :: MElasLoad

REAL(KIND=DBL), DIMENSION(ElasNdof) :: ElasDisplacement
REAL(KIND=DBL), DIMENSION(DiffNdof*2) :: DiffDisplacement
INTEGER, DIMENSION(4) ::
tempxdiff,tempydiff,tempxelas,tempyelas,tempvec2,tempRvec,tempZvec ! holder arrays for
DoF functions
REAL(KIND=DBL) :: tempscalar2 ! holder for accumulating volume vectors
INTEGER :: ierror,istat,status
!
! call value setting routines to initialize some of the above arrays
CALL setSolidElements(SolidElements)
CALL setXNodePosInit(DiffXNodePosInit,ElasXNodePosInit)
CALL setYNodePosInit(DiffYNodePosInit,ElasYNodePosInit)
CALL setElasNodePosInit(ElasXNodePosInit,ElasYNodePosInit,ElasNodePosInit)
CALL setDiffNodePosInit(DiffXNodePosInit,DiffYNodePosInit,DiffNodePosInit)
CALL setDiffDoFDirichlet(DiffDoFDirichlet)
CALL setElasDoFDirichlet(ElasDoFDirichlet)
! initializing other arrays

Phiold = PhioldInit
Phinew = 0._DBL
PhiPred = 0._DBL
FluxVelPrev = 0._DBL
FluxVelNext = 0._DBL

Disp = 0._DBL
Vel = 0._DBL
Acc = 0._DBL
AVector = 0._DBL
Aprev = 0._DBL

Dispprev = 0._DBL
Velprev = 0._DBL
Accprev = 0._DBL
Disppred = 0._DBL
Velpred = 0._DBL
Accpred = 0._DBL
Dispnext = 0._DBL
Velnext = 0._DBL
Accnext = 0._DBL


DiffNodePos = DiffNodePosInit
ElasNodePos = ElasNodePosInit
Temp = TempInit
Power = 0._DBL
LinearExpansion = Temp*9.786468D-9 + 9.918253D-6
SpecificHeat = Temp*0.118526_DBL + 104.69832_DBL
DTempDR = 0._DBL
DTempDZ = 0._DBL
ElasDens = ElasDensSolid
! set initial elasvolume vector
DO i = 1,ElasNelem
       CALL ElemVolume(ElasNodePosInit,ElasNdof,ElasXIndex,ElasYIndex,i,tempscalar2)
       InitialElasVolume(i) = tempscalar2
END DO
DO i = 1, ElasNelem
       ElasMass(i) = ElasDens(i)*InitialElasVolume(i) ! may need do loop for vector mult
END DO
ElasVolume = InitialElasVolume
! set initial diffvolume vector
DO i = 1,DiffNelem
       CALL ElemVolume(DiffNodePosInit,2*DiffNdof,DiffXIndex,DiffYindex,i,tempscalar2)
       InitialDiffVolume(i) = tempscalar2
END DO
```

```
DiffVolume = InitialDiffVolume
! conditional statement needed to set diffdens, atomicweight, and atomicnumber
DO i = 1,DiffNelem
        IF ( IsSolid(i) == 1 ) THEN
                DiffDens(i) = DiffDensSolid
                AtomicWeight(i) = 207.28789_DBL
                AtomicNumber(i) = 82._DBL
        ELSE
                DiffDens(i) = 0.00125_DBL
                AtomicWeight(i) = 14.46_DBL
                AtomicNumber(i) = 7.23_DBL
        END IF
END DO
DO i = 1, DiffNelem
        DiffMass(i) = DiffDens(i)*InitialDiffVolume(i) ! may need do loop for vector mult
END DO
Ncc = 0._DBL
SigmaFission = 0._DBL
SigmaAbsorption = 0._DBL
DiffusionLength = 0._DBL
NeutronSource = SourceTerm ! in global constants
EnormousCoefficient = 0._DBL
DiffusionConstant = 0._DBL
!KDiff = 0.
!MDiff = 0.
!FDiff = 0.
!KDiffReflected = 0.
!MDiffReflected = 0.
!FDiffReflected = 0.
!MDiffFinal = 0.
!KDiffFinal = 0.
!FDiffFinal = 0.
!Fload = 0.
!Phinew = 0.
!PhiNodalReflected = 0._DBL
PhiNodal = 0._DBL
RefTemp = 0._DBL
YoungsModulus = -1.057457D+8 * Temp + 9.5445D+10
DTempDR = 0._DBL
DTempDZ = 0._DBL
BodyForceR = 0._DBL
BodyForceZ = 0._DBL
TractionRightR = 0._DBL
TractionLeftR = 0._DBL
TractionTopZ = 0._DBL
TractionBottomZ = 0._DBL
!KElas = 0.
!MElas = 0.
!FElas = 0.
!ElasRight = 0.
!ElasLeft = 0.
!ElasTop = 0.
!KElasFinal = 0.
!MElasFinal = 0.
!KElasHat = 0.
!FElasFinal = 0.
!TempVector = 0.
ElasDisplacement = 0._DBL
DiffDisplacement = 0._DBL
tempxdiff = 0._DBL
tempydiff = 0._DBL
tempxelas = 0._DBL
tempyelas = 0._DBL
tempvec2 = 0._DBL
!
! open files for writing
```

```
OPEN(UNIT=1,FILE='FluxNodal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=2,FILE='ElasDisp.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=3,FILE='DiffDisp.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=4,FILE='TempRise.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=5,FILE='DNodePos.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=6,FILE='DiffDens.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=7,FILE='ElasDens.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
!OPEN(UNIT=8,FILE='FluxNodalRefl.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=9,FILE='KDiffFinal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=10,FILE='MDiffFinal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=11,FILE='FDiffFinal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=12,FILE='KElasFinal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=13,FILE='MElasFinal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=14,FILE='FElasFinal.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=15,FILE='DataVector.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=16,FILE='Power.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=17,FILE='ElasVolCheck.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=18,FILE='DiffVolCheck.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=19,FILE='ElasMassCheck.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)
OPEN(UNIT=20,FILE='DiffMassCheck.DAT',STATUS='REPLACE',ACTION='WRITE',IOSTAT=ierror)

! start main program block while loop
TimeChunks = 0
farout: DO
        IF ( TimeChunks > NumTimeChunks ) EXIT
        IF ( Temp(1) > 1100._DBL ) EXIT
        ! for loop structure to set parameter array values
                inner1: DO q = 1,DiffNelem
                        Ncc(q) = DiffDens(q)*(1._DBL/AtomicWeight(q))*(6.022D+23) ! atoms
per cc vector
                        IF ( IsSolid(q) == 1 ) THEN
                                SigmaFission(q) = Ncc(q)*sigmaF
                                SigmaAbsorption(q) = Ncc(q)*sigmaA
                                SigmaScatter(q) = Ncc(q)*sigmaS
                        ELSE
                                SigmaFission(q) = 0._DBL
                                SigmaAbsorption(q) = Ncc(q)*sigmaAair
                                SigmaScatter(q) = Ncc(q)*sigmaSair
                        END IF
                        DiffusionLength(q) = (1._DBL/3._DBL) / ((SigmaScatter(q))*(1._DBL-
(2._DBL/(3._DBL*AtomicNumber(q)))))
                        NeutronSource(q) = SourceTerm/DiffusionLength(q)
                        EnormousCoefficient(q) = (1._DBL/DiffusionLength(q)) *
(SigmaAbsorption(q) - FissionNubar*SigmaFission(q))
                        DiffusionConstant(q) = 1._DBL/(NeutronVelocity*DiffusionLength(q))
                END DO inner1


                !DEALLOCATE (KDiffFinal,FDiffFinal,MDiffFinal,FLoad,MLoad, STAT=status)
                ! call assembly, BC routines for diffusion solution
                ALLOCATE (KDiff(DiffNdof,DiffNdof), STAT=istat)
                CALL
MatrixAssemble(diffkelem,DiffNodePos,DiffDoF,DiffNdof,EnormousCoefficient,DiffNelem,DiffN
dof*2,4,KDiff)
                ALLOCATE (MDiff(DiffNdof,DiffNdof), STAT=istat)
                CALL
MatrixAssemble(diffmelem,DiffNodePos,DiffDoF,DiffNdof,DiffusionConstant,DiffNelem,DiffNdo
f*2,4,MDiff)
                CALL MatrixLump(MDiff,DiffNdof,MDiff)
                ALLOCATE (FDiff(DiffNdof), STAT=istat)
                CALL
VectorAssemble2(difffelem,DiffNodePos,DiffDoF,DiffNdof,NeutronSource,DiffNelem,DiffNdof*2
,4,FDiff)
                CALL DirichletApply(KDiff,DiffDoFDirichlet,DiffNdof,(DiffNdofDir),KDiff)
                CALL DirichletApply(MDiff,DiffDoFDirichlet,DiffNdof,(DiffNdofDir),MDiff)
```

```
              ALLOCATE (KDiffFinal((DiffNdof-DiffNdofDir),(DiffNdof-DiffNdofDir)) ,
STAT=istat)
              CALL MatrixConditioner(KDiff,DiffNdof,(DiffNdofDir),KDiffFinal)
              ALLOCATE (MDiffFinal((DiffNdof-DiffNdofDir),(DiffNdof-DiffNdofDir)) ,
STAT=istat)
              CALL MatrixConditioner(MDiff,DiffNdof,(DiffNdofDir),MDiffFinal)
              ALLOCATE (FDiffFinal((DiffNdof-DiffNdofDir)) , STAT=istat)
              CALL
VectorConditioner(FDiff,DiffDoFDirichlet,DiffNdof,(DiffNdofDir),FDiffFinal)
              DEALLOCATE(KDiff,MDiff,FDiff, STAT=status)
              diffsolverdim = (DiffNdof-DiffNdofDir)
              ALLOCATE (FLoad(diffsolverdim), STAT=istat)
              ALLOCATE (MLoad(diffsolverdim,diffsolverdim), STAT=istat)




              t = 0.
              step = globalstep

              inner3: DO i = 1,DiffNelem
                      CALL DiffXIndex(i,tempRvec)
                      CALL DiffYIndex(i,tempZvec)
                      ElemRCenter(i) = (DiffNodePos(tempRvec(1)) +
DiffNodePos(tempRvec(2)) + DiffNodePos(tempRvec(3)) + &
                      DiffNodePos(tempRvec(4))) / 4._DBL
                      !
                      ElemZCenter(i) = (DiffNodePos(tempZvec(1)) +
DiffNodePos(tempZvec(2)) + DiffNodePos(tempZvec(3)) + &
                      DiffNodePos(tempZvec(4))) / 4._DBL
              END DO inner3

              !IF ( TimeChunks == 0 ) THEN
              !       CALL GJSolver(MDiffFinal,FDiffFinal,diffsolverdim,FluxVelPrev)
              !END IF

              FluxVelPrev = 0._DBL

              !IF (TimeChunks == 0) THEN
              !       CALL GJSolver(KDiffFinal,FluxVelPrev,diffsolverdim,StaticCheck)
              !END IF


              !inner2: DO
              !       IF ( t > Time ) EXIT
              !       FLoad = MATMUL(MDiffFinal,Phiold) - B2*MATMUL(KDiffFinal,Phiold) +
(B1+B2)*FDiffFinal
              !       MLoad = MDiffFinal + B1*KDiffFinal
              !       CALL GJSolver(MLoad, FLoad, diffsolverdim, Phinew) ! call solver
here
              !       Phiold = Phinew
              !       CALL
DispFinalize(Phinew,DiffNdof,DiffDoFDirichlet,DiffNdofDir,PhiNodal)
              !       DO i = 1,DiffNelem
              !              CALL DiffDoF(i,tempvec2)
              !              PhiElem(i) = (1._DBL/4._DBL)*(
PhiNodal(tempvec2(1))+PhiNodal(tempvec2(2))+&
              !              PhiNodal(tempvec2(3))+PhiNodal(tempvec2(4)) )
              !       END DO
              !       DO i = 1,ElasNelem
              !              Power(i) =
SigmaFission(SolidElements(i))*PhiElem(SolidElements(i))* &
              !
       DiffVolume(SolidElements(i))*EnergyPerFission*EF
              !       END DO
```

```
!       DO i = 1,ElasNelem
!               Temp(i) = Temp(i) + ( (step*Power(i)) /
(SpecificHeat(i)*ElasMass(i)) )
!       END DO
!       t = t + step
!END DO inner2

MLoad = MDiffFinal + DiffAlpha*step*KDiffFinal

inner21: DO
        IF ( t > Time ) EXIT
        PhiPred = Phiold + (1._DBL - DiffAlpha)*step*FluxVelPrev
        FLoad = FDiffFinal - MATMUL(KDiffFinal,PhiPred)
        CALL GJSolver(MLoad,FLoad,diffsolverdim,FluxVelNext)
        Phinew = PhiPred + DiffAlpha*step*FluxVelNext
        Phiold = Phinew
        FluxVelPrev = FluxVelNext
        CALL
DispFinalize(Phinew,DiffNdof,DiffDoFDirichlet,DiffNdofDir,PhiNodal)
        DO i = 1,DiffNelem
                CALL DiffDoF(i,tempvec2)
                PhiElem(i) = (1._DBL/4._DBL)*(
PhiNodal(tempvec2(1))+PhiNodal(tempvec2(2))+&
                PhiNodal(tempvec2(3))+PhiNodal(tempvec2(4)) )
        END DO
        DO i = 1,ElasNelem
                Power(i) =
SigmaFission(SolidElements(i))*PhiElem(SolidElements(i))* &

       DiffVolume(SolidElements(i))*EnergyPerFission
        END DO
        DO i = 1,ElasNelem
                Temp(i) = Temp(i) + ( (step*Power(i)*EF) /
(SpecificHeat(i)*ElasMass(i)) )
        END DO
        t = t + step
END DO inner21

DEALLOCATE (KDiffFinal,FDiffFinal,MDiffFinal,FLoad,MLoad, STAT=status)
CALL DispFinalize(Phinew,DiffNdof,DiffDoFDirichlet,DiffNdofDir,PhiNodal)
WRITE(1,100) PhiNodal
100 FORMAT (1X,1000E14.5)
inner4: DO i = 1,ElasNelem
        !
        qq = SolidElements(i)
        ElemRCenter(qq) = ElemRCenter(qq) / 100._DBL
        ElemZCenter(qq) = ElemZCenter(qq) / 100._DBL
        RefTemp(i) = Temp(i) - 25.
        !
END DO inner4
inner41: DO i = 1,ElasNelem
        qq = SolidElements(i)
        IF ( ElasCol(i) == 1 ) THEN
                DTempDR(i) = (RefTemp(i)-RefTemp(i+1))/(ElemRCenter(qq)-
ElemRCenter(qq+1))
        ELSE IF ( ElasCol(i) == ElasNCols ) THEN
                DTempDR(i) = (RefTemp(i)-RefTemp(i-1))/(ElemRCenter(qq)-
ElemRCenter(qq-1))
        ELSE
                DTempDR(i) = (RefTemp(i+1)-RefTemp(i-
1))/(ElemRCenter(qq+1)-ElemRCenter(qq-1))
        END IF
        !
        IF ( ElasRow(i) == 1 ) THEN
                DTempDZ(i) = (RefTemp(i)-
RefTemp(i+ElasNCols))/(ElemZCenter(qq)-ElemZCenter(qq+DiffNCols))
```

```
                        ELSE IF ( ElasRow(i) == ElasNRows ) THEN
                                DTempDZ(i) = (RefTemp(i)-RefTemp(i-
ElasNCols))/(ElemZCenter(qq)-ElemZCenter(qq-DiffNCols))
                        ELSE
                                DTempDZ(i) = (RefTemp(i+ElasNCols)-RefTemp(i-
ElasNCols))/(ElemZCenter(qq+DiffNCols)-ElemZCenter(qq-DiffNCols))
                        END IF
                END DO inner41
                inner5: DO i = 1,ElasNelem
                        YoungsModulus(i) = -1.057457D+8 * Temp(i) + 9.5445D+10
                        LinearExpansion(i) = Temp(i)*9.786468D-9 + 9.918253D-6
                        SpecificHeat(i) = Temp(i)*0.118526_DBL + 104.69832_DBL
                        BodyForceR(i) = -(1._DBL/(1._DBL-2._DBL*Poisson)) *
LinearExpansion(i)*YoungsModulus(i)*DTempDR(i)
                        BodyForceZ(i) = -ZtracSwitch*(1._DBL/(1._DBL-2._DBL*Poisson)) *
LinearExpansion(i)*YoungsModulus(i)*DTempDZ(i)
                        TractionRightR(i) = (1._DBL/(1._DBL-2._DBL*Poisson)) *
LinearExpansion(i)*YoungsModulus(i)*RefTemp(i)
                        TractionLeftR(i) =  -(1._DBL/(1._DBL-2._DBL*Poisson)) *
LinearExpansion(i)*YoungsModulus(i)*RefTemp(i)
                        TractionTopZ(i) = ZtracSwitch*(1._DBL/(1._DBL-2._DBL*Poisson)) *
LinearExpansion(i)*YoungsModulus(i)*RefTemp(i)
                        TractionBottomZ(i) = -ZtracSwitch*(1._DBL/(1._DBL-2._DBL*Poisson))
* LinearExpansion(i)*YoungsModulus(i)*RefTemp(i)
                END DO inner5
                WRITE(4,200) RefTemp
                200 FORMAT (1X,1000E14.5)
                WRITE(16,1600) Power
                1600 FORMAT (1X,1000E14.5)

                DEALLOCATE(ElasRight,ElasLeft,ElasTop,ElasBottom, STAT=status)
                !DEALLOCATE(KElasFinal,STAT=status)
                !DEALLOCATE(TempVector,KElasHat,MElasFinal,FElasFinal, STAT=status)
                ALLOCATE (KElas(ElasNdof,ElasNdof), STAT=istat)
                CALL
StiffnessMatrixAssemble(elaskelem,ElasNodePos,ElasDoF,ElasNdof,YoungsModulus,ElasNelem,El
asNdof,8,KElas)
                ALLOCATE (MElas(ElasNdof,ElasNdof), STAT=istat)
                CALL
StiffnessMatrixAssemble(elasmelem,ElasNodePos,ElasDoF,ElasNdof,ElasDens,ElasNelem,ElasNdo
f,8,MElas)
                ALLOCATE (FElas(ElasNdof), STAT=istat)
                Call
VectorAssemble(elasfelem,ElasNodePos,ElasDoF,ElasNdof,BodyForceR,BodyForceZ,ElasNelem,Ela
sNdof,8,FElas)
                ALLOCATE (ElasRight(ElasNdof), STAT=istat)
                Call
BoundaryAssembleRight(elasrightelem,ElasNodePos,ElasDoF,ElasNdof,TractionRightR,ElasNelem
,ElasNdof,8,ElasRight)
                ALLOCATE (ElasLeft(ElasNdof), STAT=istat)
                Call
BoundaryAssembleLeft(elasleftelem,ElasNodePos,ElasDoF,ElasNdof,TractionLeftR,ElasNelem,El
asNdof,8,ElasLeft)
                ALLOCATE (ElasTop(ElasNdof), STAT=istat)
                Call
BoundaryAssembleTop(elastopelem,ElasNodePos,ElasDoF,ElasNdof,TractionTopZ,ElasNelem,ElasN
dof,8,ElasTop)
                ALLOCATE (ElasBottom(ElasNdof), STAT=istat)
                Call
BoundaryAssembleBottom(elasbottomelem,ElasNodePos,ElasDoF,ElasNdof,TractionBottomZ,ElasNe
lem,ElasNdof,8,ElasBottom)
                FElas = ElasRight + ElasLeft + ElasTop + ElasBottom + FElas
                !DEALLOCATE(ElasRight,ElasLeft,ElasTop,ElasBottom, STAT=status)
                CALL MatrixLump(MElas,ElasNdof,MElas)
                CALL DirichletApply(KElas,ElasDoFDirichlet,ElasNdof,(ElasNdofDir),KElas)
                CALL DirichletApply(MElas,ElasDoFDirichlet,ElasNdof,(ElasNdofDir),MElas)
```

```
                ALLOCATE(FElasFinal(ElasNdof-(ElasNdofDir)), STAT=istat)
                CALL
VectorConditioner(FElas,ElasDoFDirichlet,ElasNdof,(ElasNdofDir),FElasFinal)
                DEALLOCATE(FElas, STAT=status)
                ALLOCATE(KElasFinal((ElasNdof-(ElasNdofDir)),(ElasNdof-(ElasNdofDir))),
STAT=istat)
                CALL MatrixConditioner(KElas,ElasNdof,(ElasNdofDir),KElasFinal)
                DEALLOCATE(KElas, STAT=status)
                ALLOCATE(MElasFinal(ElasNdof-(ElasNdofDir),ElasNdof-(ElasNdofDir)) ,
STAT=istat)
                CALL MatrixConditioner(MElas,ElasNdof,(ElasNdofDir),MElasFinal)
                DEALLOCATE(MElas, STAT=status)
                !ALLOCATE(KElasHat(ElasNdof-(ElasNdofDir),ElasNdof-(ElasNdofDir)),
STAT=istat)
                !KElasHat = KElasFinal + a3*MElasFinal
                elassolverdim = ElasNdof - (ElasNdofDir)

                IF (TimeChunks == NumTimeChunks) THEN
                        CALL GJSolver(KElasFinal,FElasFinal,elassolverdim,StaticCheck)
                END IF


                !DEALLOCATE(KElasFinal,STAT=status)
                ! start while loop structure for elasticity differencing
                ALLOCATE(TempVector(ElasNdof-(ElasNdofDir)),STAT=istat)
                t = 0.
                step = timestep_elas
                !Disp = 0._DBL
                !Dispprev = 0._DBL
                !Acc = 0._DBL
                !Accprev = 0._DBL
                !inner6: DO
                !       IF ( t > Time ) EXIT
                !       Velprev = Vel
                !       Accprev = Acc
                !       Dispprev = Disp
                !       Aprev = AVector
                !       TempVector = FElasFinal + MATMUL(MElasFinal,Aprev)
                !       CALL GJSolver(KElasHat, TempVector, elassolverdim, Disp) ! call
solver
                !       Acc = a3*(Disp-Dispprev) - a4*Velprev - a5*Accprev
                !       Vel = Velprev + a2*Accprev + a1*Acc
                !       AVector = a3*Disp + a4*Vel + a5*Acc
                !       t = t + step
                !END DO inner6

                ALLOCATE(MElasLoad(elassolverdim,elassolverdim),STAT=istat)
                ALLOCATE(FElasLoad(elassolverdim),STAT=istat)

                !IF ( TimeChunks == 0 ) THEN
                !       CALL GJSolver(MElasFinal,FElasFinal,elassolverdim,Accprev)
                !END IF

                Accprev = 0._DBL

                MElasLoad = MElasFinal + Beta*(step**2._DBL)*KElasFinal

                inner61: DO
                        IF ( t > Time ) EXIT
                                Disppred = Dispprev + step*Velprev +
(step**2._DBL)*(0.5_DBL)*(1._DBL-2._DBL*Beta)*Accprev
                                Velpred = Velprev + (1-Gamma)*step*Accprev
                                FElasLoad = FElasFinal - MATMUL(KElasFinal,Disppred)
                                CALL GJSolver(MElasLoad,FElasLoad,elassolverdim,Accnext)
                                Dispnext = Disppred + Beta*(step**2._DBL)*Accnext
                                Velnext = Velpred + Gamma*step*Accnext
```

```
                                Dispprev = Dispnext
                                Velprev = Velnext
                                Accprev = Accnext
                                t = t + step
                END DO inner61

                DEALLOCATE(MElasLoad,FElasLoad,STAT=status)


                DEALLOCATE(KElasFinal,STAT=status)
                DEALLOCATE(TempVector,KElasHat,MElasFinal,FElasFinal, STAT=status)
                !CALL
DispFinalize(Disp,ElasNdof,ElasDoFDirichlet,(ElasNdofDir),ElasDisplacement)
                CALL
DispFinalize(Dispnext,ElasNdof,ElasDoFDirichlet,(ElasNdofDir),ElasDisplacement)
                WRITE(2,300) ElasDisplacement
                300 FORMAT (1X,1000E14.5)

                DO i = 1,ElasNdof
                        ElasNodePos(i) = ElasNodePosInit(i) + ElasDisplacement(i)
                END DO

                inner7: DO i = 1,ElasNelem
                CALL DiffXIndex(SolidElements(i),tempxdiff)
                CALL DiffYIndex(SolidElements(i),tempydiff)
                CALL ElasXIndex(i,tempxelas)
                CALL ElasYIndex(i,tempyelas)
                        farinner1: DO s = 1,4
                                DiffDisplacement(tempxdiff(s)) =
100._DBL*ElasDisplacement(tempxelas(s))
                                DiffDisplacement(tempydiff(s)) =
100._DBL*ElasDisplacement(tempyelas(s))
                        END DO farinner1
                END DO inner7
                WRITE(3,400) DiffDisplacement
                400 FORMAT (1X,1500E14.5)
                inner8: DO q = 1,2*DiffNdof
                        DiffNodePos(q) = DiffNodePosInit(q) + DiffDisplacement(q)
                END DO inner8

                ! repositioning boundary nodes to retain correct spacing

                inner81: DO q = 1,DiffNelem
                        IF ( DiffRow(q) == DiffNRows ) THEN
                                CALL DiffYIndex(q,tempydiff)
                                DiffNodePos(tempydiff(4)) = DiffNodePos(tempydiff(1)) +
2.13_DBL*DiffConstantInit
                                DiffNodePos(tempydiff(3)) = DiffNodePos(tempydiff(2)) +
2.13_DBL*DiffConstantInit
                        ELSE IF ( DiffCol(q) == DiffNCols ) THEN
                                CALL DiffXIndex(q,tempxdiff)
                                DiffNodePos(tempxdiff(2)) = DiffNodePos(tempxdiff(1)) +
2.13_DBL*DiffConstantInit
                                DiffNodePos(tempxdiff(3)) = DiffNodePos(tempxdiff(4)) +
2.13_DBL*DiffConstantInit
                        END IF
                END DO inner81

                WRITE(5,500) DiffNodePos
                500 FORMAT (1X,1500E14.5)
                inner9: DO q = 1,DiffNelem
                        CALL
ElemVolume(DiffNodePos,2*DiffNdof,DiffXIndex,DiffYIndex,q,tempscalar2)
                        DiffVolume(q) = tempscalar2
                        DiffDens(q) = DiffMass(q)/DiffVolume(q)
                END DO inner9
```

```
               WRITE(6,600) DiffDens
               600 FORMAT (1X,1000E14.5)
               inner10: DO q = 1,ElasNelem
                        CALL
ElemVolume(ElasNodePos,ElasNdof,ElasXIndex,ElasYIndex,q,tempscalar2)
                        ElasVolume(q) = tempscalar2
                        ElasDens(q) = ElasMass(q)/ElasVolume(q)
               END DO inner10

               WRITE(7,700) ElasDens
               700 FORMAT (1X,1000E14.5)
               WRITE(17,700) ElasVolume

               WRITE(18,700) DiffVolume

               WRITE(19,700) ElasMass

               WRITE(20,700) DiffMass


       TimeChunks = TimeChunks + 1
END DO farout

!
! note that the size integers in the matrix outputs below must be
! changed if the mesh size is changed
!

WRITE(9,900) KDiffFinal
900 FORMAT (1X, 30E13.3)
!
WRITE(10,900) MDiffFinal
!
!
WRITE(12,1000) KElasHat
1000 FORMAT (1X, 15E13.3)
!
WRITE(13,1000) MElasFinal
!
!
WRITE(11,1100) FDiffFinal
!
!
WRITE(14,1100) FElasFinal

WRITE(15,1100) ElasLeft
1100 FORMAT (1X, E13.4)



END PROGRAM testmain
```

Appendix D: Documentation for 2D Code

This appendix contains the documentation for the source code in Appendix C. Explanations of program modules, subroutines, functions and parameters may be found here. The two-dimensional code shall be referred to as *TDMain* in this appendix.

TDMain as seen in Appendix C is a pre-alpha stage code. It is assumed the user has access to the source code and a Fortran 95 compiler. The code is used by modifying parameters listed in the first program module, recompiling, and running the resulting executable.

D.1 Parameter List and Functionality

- *DBL*: controls the precision of all real variables in the code. Default is 8 for double precision;
- *PI*: Value of pi. Set to 3.1415926535;
- *ElasDensSolid*: Initial density of solid region in kg/m^3;
- *ZtracSwitch* : Switch to turn axial traction and body forces on and off. A value of 1 turns these forces on, a value of 0 turns them off. Other values are invalid;
- *PhioldInit*: Initial value for scalar flux field;
- *TempInit*: Initial value for temperature field in the solid region;
- *SourceTerm*: Source term in neutrons per second per centimeter squared in the solid region;
- *sigmaF*: Microscopic fission cross-section for solid region in cm^2;
- *sigmaA*: Microscopic absorption cross-section for solid region in cm2;
- *sigmaAair*: Microscopic absorption cross-section for air/vacuum region in cm2;

- *sigmaS*: Microscopic scattering cross-section for solid region in cm2;
- *sigmaSair*: Microscopic scattering cross-section for air/vacuum region in cm2;
- *NeutronVelocity*: Neutron velocity in cm/s;
- *FissionNubar*: Average number of neutrons produced per fission reaction;
- *EF*: Engineering factor controlling proportion of energy deposited locally as heat. Default value is 1;
- *EnergyPerFission*: Amount of energy in joules per fission reaction;
- *Poisson*: Poisson's ratio for solid material;
- *NumTimeChunks*: Number of time loops per program run;
- *Time*: Duration of time loops;
- *globalstep*: Size of time step during diffusion time marching;
- *timestep_elas*: Size of time step during elasticity time marching;
- *Gamma*: Parameter for elasticity time marching;
- *Beta*: Parameter for elasticity time marching;
- *DiffAlpha*: Parameter for diffusion time marching;
- *DiffConstantInit*: Initial value for diffusion constant, used to set boundary positions of exterior air/vacuum elements;
- *DiffNCols*: Number of columns in diffusion mesh;
- *DiffNRows*: Number of rows in diffusion mesh;
- *ElasHeight*: Height in meters of solid region;
- *ElasIR*: Inner radius in meters of hollow cylindrical solid region;
- *ElasOR*: Outer radius in meters of hollow cylindrical solid region;
- *ZClamped*: Switch turning axial displacement at the top of the cylinder on or off. Default value is 1, leaving axial displacement on. Value of 0 turns it off. All other values are invalid.

D.2 MODULE LIST

Here is a list of modules contained in program *TDMain* and a brief description of their functionality.

- **TDGlobalConstants:** Module where all control of the program takes place. Parameters controlling the functionality of the code reside in this module;
- **TDRowCol:** Contains indexing functions giving row and column position in either the elasticity or diffusion mesh solely as a function of element number;
- **TDGlobalRoutines:** Miscellaneous set of subroutines used throughout the code;
- **TDValueSetRoutines:** Set of subroutines used to initialize vectors used in the main program block;
- **TDSolverRoutines1:** Contains preliminary solver routines used in TDSolverRoutines2 and TDGaussSolver;
- **TDSolverRoutines2:** Contains final solver routines used in TDGaussSolver;
- **TDGaussSolver:** Implements the Gauss-Jordan solver by calling routines defined in the previous solver routine modules;
- **TDGlobalFunctions:** Contains shape and derivative functions, as well as indexing functions essential to the finite element implementation in later modules;
- **TDTransform1:** Contains functions defining coordinates and coordinate derivatives. First stage of coordinate transform;
- **TDJacobian:** Contains Jacobian and arctransform functions. Second stage of coordinate transform;
- **TDTransform2:** Contains further derivative functions based off definitions in previous transform modules. Third stage of coordinate transform;

- **TDTransform3:** Contains final derivatives needed for coordinate transform. Completes that process;
- **TDShapeFunctionMatrix:** Contains subroutines generating shape function vectors and matrices needed for elasticity integral evaluation;
- **TDArgMatrix:** Contains subroutine generating the 8x8 argument matrix needed for evaluating elasticity element stiffness matrices;
- **TDIntArguments:** Contains subroutines generating the complete integrands for all stiffness, mass and load calculations;
- **TDIntegrationRoutines:** Contains routines implementing Gaussian quadrature over a quadrilateral master element. Routines generally differ in arguments taken and number of integrals evaluated per loop iteration;
- **TDPreAssembly:** Contains routines evaluating individual integral contributions to the global matrices and vectors intrinsic to the final linear system of equations;
- **TDAssemblyBC:** Contains routines assembling the integrals evaluated in the pre-assembly process into correctly structured stiffness and mass matrices, and load vectors.

D.3 PROCEDURE LIST AND FUNCTIONALITY

Here we list each procedure -- functions and subroutines -- contained in the code *TDMain*. A description of the arguments accepted, procedure functionality and procedure output follows.

Function: DiffRow

Accepts: element number

Functionality: Simple mesh location function

Returns: Row location in diffusion mesh

Function: DiffCol

Accepts: element number

Functionality: Simple mesh location function

Returns: Column location in diffusion mesh

Function: ElasRow

Accepts: element number

Functionality: Simple mesh location function

Returns: Row location in elasticity mesh

Function: ElasCol

Accepts: element number

Functionality: Simple mesh location function

Returns: Column location in diffusion mesh

Subroutine: VectorSort

Accepts: input vector, dummy variable for output vector

Functionality: Sorts vector entries in ascending order

Returns: Sorted vector

Function: IsSolid

Accepts: element number

Functionality: Checks whether element in diffusion mesh is in the solid region

Returns: 1 if element is solid, 0 if element is air or vacuum

Subroutine: setSolidElements

Accepts: dummy variable for output vector

Functionality: Generates values for SolidElements vector in main program block

Returns: vector containing element numbers of solid elements in the diffusion mesh

Subroutine: setXNodePosInit

Accepts: dummy variable for output vector 1, dummy variable for output vector 2

Functionality: Generates vectors containing initial radial coordinates at mesh nodes for the diffusion mesh and the elasticity mesh

Returns: vector containing radial positions for diffusion, vector containing radial positions for elasticity

Subroutine: setYNodePosInit

Accepts: dummy variable for output vector 1, dummy variable for output vector 2

Functionality: Generates vectors containing initial axial coordinates at mesh nodes for the diffusion mesh and the elasticity mesh

Returns: vector containing axial positions for diffusion, vector containing axial positions for elasticity

Subroutine: setElasNodePosInit

Accepts: input vector 1, input vector 2, dummy variable for output vector

Functionality: Accepts radial and axial position vectors for the elasticity mesh and combines them into one vector containing all the node positions

Returns: Single node position vector for elasticity mesh

Subroutine: setDiffNodePosInit

Accepts: input vector 1, input vector 2, dummy variable for output vector

Functionality: Accepts radial and axial position vectors for the diffusion mesh and combines them into one vector containing all the node positions

Returns: Single node position vector for diffusion mesh

Subroutine: setDiffDoFDirichlet

Accepts: dummy variable for output vector

Functionality: populates vector with Dirichlet condition nodes for the diffusion mesh

Returns: Vector containing Dirichlet nodes for the diffusion mesh

Subroutine: setElasDoFDirichlet

Accepts: dummy variable for output vector

Functionality: populates vector with Dirichlet condition nodes for the elasticity mesh

Returns: Vector containing Dirichlet nodes for the elasticity mesh

Subroutine: setCoeffMat

Accepts: dummy variable for output matrix

Functionality: populates 4x4 matrix with appropriate material coefficients for linear axisymmetric elasticity

Returns: 4x4 coefficient matrix for linear axisymmetric elasticity

Subroutine: Augmented

Accepts: Input matrix, input vector, dimension of square matrix and vector, dummy argument for output matrix

Functionality: Augments the matrix with the input vector.

Returns: Augmented matrix.

Subroutine: RowSwitch

Accepts: Augmented matrix, number of rows in matrix, two rows whose locations in the matrix are to be switched, dummy argument for output matrix

Functionality: Switches the position of two rows in an augmented matrix

Returns: Matrix of the same size and dimension as the input matrix, but with row 1 and row 2 switched.

Subroutine: GetDivFactor

Accepts: Augmented matrix, number of rows in matrix, pivot row number, column number, dummy argument for output vector

Functionality: Accepts the current pivot row and column number in a forward elimination process, and solves for the division factor of each subsequent row after the pivot row.

Returns: Vector of division factors for forward elimination.

Subroutine: SearchForPivot

Accepts: Augmented matrix, number of rows in matrix, row number, column number, dummy argument for output integer

Functionality: Searches the augmented matrix starting at row number and column number for a nonzero entry in that column. It designates the row number of that nonzero entry as the pivot row.

Returns: Pivot row integer for use in forward elimination

Subroutine: ForwardElim

Accepts: Augmented matrix, number of rows in matrix, dummy argument for output matrix

Functionality: Performs a standard forward elimination process on an augmented matrix.

Returns: Upper diagonal matrix ready for backward substitution.

Subroutine: BackSub

Accepts: Forward eliminated matrix, number of rows in matrix, dummy argument for output matrix

Functionality: Performs standard backward substitution on a forward eliminated matrix.

Returns: Augmented matrix in full diagonal form, with solutions to equations occupying the last column.

Subroutine: GJSolver

Accepts: Square matrix of equation coefficients, vector of loads, number of rows in matrix, dummy argument for output vector

Functionality: Calls Augment, ForwardElim, and BackSub in succession.

Returns: The last column of the backward substituted augmented matrix. This is the vector of solutions to the linear system of equations.

Function: Psi

Accepts: shape function number, master element Xi coordinate, master element Eta coordinate

Functionality: Generates value of linear shape function on the master element

Returns: Linear shape function value on the master element at a particular point (Xi, Eta)

Function: dPsidEta

Accepts: shape function number, master element Xi coordinate

Functionality: Generates value of the derivative of a linear shape function on the master element with respect to Eta

Returns: Linear shape function derivative with respect to Eta value on the master element at a particular value of Xi

Function: dPsidXi

Accepts: shape function number, master element Eta coordinate

Functionality: Generates value of the derivative of a linear shape function on the master element with respect to Xi

Returns: Linear shape function derivative with respect to Xi value on the master element at a particular value of Eta

Subroutine: ElasDoF

Accepts: element number, dummy variable for 8x1 output index vector

Functionality: Accepts element number on the elasticity mesh and returns the eight nodal degrees of freedom for that element, in order of shape function number

Returns: 8x1 vector giving nodal degree of freedom numbers for a particular elasticity element

Subroutine: DiffDoF

Accepts: element number, dummy variable for 4x1 output index vector

Functionality: Accepts element number on the diffusion mesh and returns the four nodal degrees of freedom for that element, in order of shape function number

Returns: 4x1 vector giving nodal degree of freedom numbers for a particular diffusion element

Subroutine: DiffXIndex

Accepts: element number, dummy variable for 4x1 output index vector

Functionality: Accepts element number on the diffusion mesh and returns the four vector addresses in the nodal position vector for the radial coordinates of the four element nodes

Returns: 4x1 vector giving the addresses of the radial coordinates of the element nodes in the global diffusion node position vector

Subroutine: DiffYIndex

Accepts: element number, dummy variable for 4x1 output index vector

Functionality: Accepts element number on the diffusion mesh and returns the four vector addresses in the nodal position vector for the axial coordinates of the four element nodes

Returns: 4x1 vector giving the addresses of the axial coordinates of the element nodes in the global diffusion node position vector

Subroutine: ElasXIndex

Accepts: element number, dummy variable for 4x1 output index vector

Functionality: Accepts element number on the elasticity mesh and returns the four vector addresses in the nodal position vector for the radial coordinates of the four element nodes

Returns: 4x1 vector giving the addresses of the radial coordinates of the element nodes in the global elasticity node position vector

Subroutine: ElasYIndex

Accepts: element number, dummy variable for 4x1 output index vector

Functionality: Accepts element number on the elasticity mesh and returns the four vector addresses in the nodal position vector for the axial coordinates of the four element nodes

Returns: 4x1 vector giving the addresses of the axial coordinates of the element nodes in the global elasticity node position vector

Function: X

Accepts: node position vector, node position vector dimension, index generation subroutine, element number, Xi coordinate, Eta coordinate

Functionality: Accesses node position vector using the generated local index, then uses local shape functions to interpolate the global radial coordinate at position (Xi, Eta) on the master element

Returns: Global radial coordinate value

Function: Y

Accepts: node position vector, node position vector dimension, index generation subroutine, element number, Xi coordinate, Eta coordinate

Functionality: Accesses node position vector using the generated local index, then uses local shape functions to interpolate the global axial coordinate at position (Xi, Eta) on the master element

Returns: Global axial coordinate value

Function: dXdXi

Accepts: node position vector, node position vector dimension, index generation subroutine, element number, Eta coordinate

Functionality: Accesses node position vector using the generated local index, then uses local shape functions to interpolate the derivative of the global radial coordinate with respect to Xi at position (*, Eta) on the master element

Returns: Value for derivative of the global radial coordinate with respect to Xi

Function: dYdXi

Accepts: node position vector, node position vector dimension, index generation subroutine, element number, Eta coordinate

Functionality: Accesses node position vector using the generated local index, then uses local shape functions to interpolate the derivative of the global axial coordinate with respect to Xi at position (*, Eta) on the master element

Returns: Value for derivative of the global axial coordinate with respect to Xi

Function: dXdEta

Accepts: node position vector, node position vector dimension, index generation subroutine, element number, Xi coordinate

Functionality: Accesses node position vector using the generated local index, then uses local shape functions to interpolate the derivative of the global radial coordinate with respect to Eta at position (Xi, *) on the master element

Returns: Value for derivative of the global radial coordinate with respect to Eta

Function: dYdEta

Accepts: node position vector, node position vector dimension, index generation subroutine, element number, Xi coordinate

Functionality: Accesses node position vector using the generated local index, then uses local shape functions to interpolate the derivative of the global axial coordinate with respect to Eta at position (Xi, *) on the master element

Returns: Value for derivative of the global axial coordinate with respect to Eta

Function: Jacobian

Accepts: node position vector, node position vector dimension, index generation subroutine 1, index generation subroutine 2, element number, Xi coordinate, Eta coordinate

Functionality: Accesses node position vector using the generated local index, then uses previously defined functions to calculate the value of the Jacobian for the local element at master element coordinates (Xi, Eta)

Returns: Value for element Jacobian at position (Xi, Eta)

Function: ArcTransform*

Accepts: node position vector, node position vector dimension, index generation subroutine 1, index generation subroutine 2, element number

Functionality: Accesses node position vector using the generated local index, then uses previously defined functions to calculate the value of the arc length of the applicable element side. Used in applying traction boundary conditions.

Returns: Value for the arc length of the appropriate side of a boundary element

Function: dXidX

Accepts: node position vector, node position vector dimension, index generation subroutine 1, index generation subroutine 2, element number, Xi coordinate, Eta coordinate

Functionality: Calculates the derivative of the master element coordinate Xi with respect to the global radial coordinate X. This is a simple calculation using previously defined functions and the chain rule.

Returns: Derivative of Xi with respect to X at coordinate (Xi, Eta) on the specified element

Function: dXidY

Accepts: node position vector, node position vector dimension, index generation subroutine 1, index generation subroutine 2, element number, Xi coordinate, Eta coordinate

Functionality: Calculates the derivative of the master element coordinate Xi with respect to the global axial coordinate Y. This is a simple calculation using previously defined functions and the chain rule.

Returns: Derivative of Xi with respect to Y at coordinate (Xi, Eta) on the specified element

Function: dEtadX

Accepts: node position vector, node position vector dimension, index generation subroutine 1, index generation subroutine 2, element number, Xi coordinate, Eta coordinate

Functionality: Calculates the derivative of the master element coordinate Eta with respect to the global radial coordinate X. This is a simple calculation using previously defined functions and the chain rule.

Returns: Derivative of Eta with respect to X at coordinate (Xi, Eta) on the specified element

Function: dEtadY

Accepts: node position vector, node position vector dimension, index generation subroutine 1, index generation subroutine 2, element number, Xi coordinate, Eta coordinate

Functionality: Calculates the derivative of the master element coordinate Eta with respect to the global axial coordinate Y. This is a simple calculation using previously defined functions and the chain rule.

Returns: Derivative of Eta with respect to Y at coordinate (Xi, Eta) on the specified element

Function: dPsidX

Accepts: node position vector, node position vector dimension, index generation subroutine 1, index generation subroutine 2, element number, Xi coordinate, Eta coordinate

Functionality: Calculates the derivative of the global basis function with respect to the global radial coordinate X. This uses previously defined functions and the chain rule.

Returns: Derivative of Psi with respect to X at coordinate (Xi, Eta) on the specified element

Function: dPsidY

Accepts: node position vector, node position vector dimension, index generation subroutine 1, index generation subroutine 2, element number, Xi coordinate, Eta coordinate

Functionality: Calculates the derivative of the global basis function with respect to the global axial coordinate Y. This uses previously defined functions and the chain rule.

Returns: Derivative of Psi with respect to Y at coordinate (Xi, Eta) on the specified element

Function: hVector

Accepts: coordinate Xi, coordinate Eta

Functionality: Populates the 2x8 matrix of shape functions used in integral calculations for axisymmetric linear elasticity. Used in constructing the stiffness, mass and load integrands for elasticity calculations. Not used in diffusion.

Returns: 2x8 matrix of shape functions for linear elasticity calculations.

Function: hVector****

Accepts: coordinate Xi, coordinate Eta

Functionality: Populates the 2x8 matrix of shape functions used in integral calculations for axisymmetric linear elasticity. Used in constructing the load integrands for elasticity calculations on boundary elements with traction conditions present. Not used in diffusion.

Returns: 2x8 matrix of shape functions for linear elasticity calculations on boundary elements.

Function: dhMatrix

Accepts: node position vector, node position vector dimension, index subroutine 1, index subroutine 2, element number, coordinate Xi, coordinate Eta

Functionality: Populates the 4x8 matrix created by operating on the transpose of hVector with a differential operator specific to the axisymmetric case. Appears in elasticity stiffness calculations only.

Returns: 4x8 matrix required for elasticity stiffness calculations

Subroutine: ArgumentMatrix

Accepts: node position vector, node position vector dimension, index subroutine 1, index subroutine 2, material data vector, element number, coordinate Xi, coordinate Eta, dummy variable for output matrix

Functionality: Uses previously defined function dhMatrix and its transpose, as well as local material data, to define the elasticity stiffness integrand completely for a particular element.

Returns: 8x8 matrix of element stiffness integrands

Subroutine: ElasKIntArgMatrix

Accepts: node position vector, node position vector dimension, index subroutine 1, index subroutine 2, material data vector, element number, coordinate Xi, coordinate Eta, dummy variable for output matrix

Functionality: Establishes subroutine for generating values of the elasticity stiffness integrand. Essentially the same as ArgumentMatrix except multiplied by the 2 Pi X value required for axisymmetric calculations.

Returns: 8x8 matrix of element stiffness integrands

Subroutine: ElasMIntArgMatrix

Accepts: node position vector, node position vector dimension, index subroutine 1, index subroutine 2, material data vector, element number, coordinate Xi, coordinate Eta, dummy variable for output matrix

Functionality: Establishes subroutine for generating values of the elasticity mass integrand.

Returns: 8x8 matrix of element mass integrands

Subroutine: ElasFIntArgVector

Accepts: node position vector, node position vector dimension, index subroutine 1, index subroutine 2, material data vector 1, material data vector 2, element number, coordinate Xi, coordinate Eta, dummy variable for output vector

Functionality: Establishes subroutine for generating values of the elasticity load integrand.

Returns: 8x1 vector of element load integrands

Subroutine: ElasTrac****IntArg

Accepts: node position vector, node position vector dimension, index subroutine 1, index subroutine 2, material data vector 1, element number, coordinate Xi, coordinate Eta, dummy variable for output vector

Functionality: Establishes subroutine for generating values of the elasticity traction integrand. These are nonzero only on boundary elements with traction conditions.

Returns: 8x1 vector of element traction integrands

Subroutine: VolArg

Accepts: node position vector, node position vector dimension, index subroutine 1, index subroutine 2, element number, coordinate Xi, coordinate Eta, dummy variable for output scalar

Functionality: Establishes subroutine for generating values of the volume calculation integrand.

Returns: Scalar valued integrand ready for volume calculation

Subroutine: DiffMIntArgMatrix

Accepts: node position vector, node position vector dimension, index subroutine 1, index subroutine 2, material data, element number, shape function 1, shape function 2, coordinate Xi, coordinate Eta, dummy variable for output scalar

Functionality: Establishes subroutine for generating values of the diffusion mass calculation integrand. Diffusion calculations are carried out entry by entry, so results are scalars instead of matrices and vectors.

Returns: Scalar valued integrand ready for diffusion mass calculations

Subroutine: DiffKIntArgMatrix

Accepts: node position vector, node position vector dimension, index subroutine 1, index subroutine 2, material data, element number, shape function 1, shape function 2, coordinate Xi, coordinate Eta, dummy variable for output scalar

Functionality: Establishes subroutine for generating values of the diffusion stiffness calculation integrand. Diffusion calculations are carried out entry by entry, so results are scalars instead of matrices and vectors.

Returns: Scalar valued integrand ready for diffusion stiffness calculations

Subroutine: DiffFIntArgVector

Accepts: node position vector, node position vector dimension, index subroutine 1, index subroutine 2, material data, element number, shape function number, coordinate Xi, coordinate Eta, dummy variable for output scalar

Functionality: Establishes subroutine for generating values of the diffusion load calculation integrand. Diffusion calculations are carried out entry by entry, so results are scalars instead of matrices and vectors.

Returns: Scalar valued integrand ready for diffusion load calculations

Subroutine: GaussIntegrator

Accepts: node position vector, index subroutine 1, index subroutine 2, material data, element number, shape function number 1, shape function number 2, function, node position vector dimension, material data vector dimension, dummy variable for output scalar

Functionality: Integrates function over square master element using a two-point Gaussian quadrature rule.

Returns: Scalar value ready for insertion into diffusion matrix.

Subroutine: VolumeIntegrator

Accepts: node position vector, index subroutine 1, index subroutine 2, element number, function, node position vector dimension, dummy variable for output scalar

Functionality: Integrates function over square master element using a two-point Gaussian quadrature rule. This particular routine is used for acquiring the volume of an element for density calculations.

Returns: Scalar value of element volume

Subroutine: GaussIntegrator2

Accepts: node position vector, index subroutine 1, index subroutine 2, material data, element number, shape function number 1, function, node position vector dimension, material data vector dimension, dummy variable for output scalar

Functionality: Integrates function over square master element using a two-point Gaussian quadrature rule. Same as GaussIntegrator, but uses only one shape function number. Used for diffusion load vector calculations.

Returns: Scalar valued integrand ready for insertion into diffusion load vector.

Subroutine: GaussIntegrator3

Accepts: node position vector, index subroutine 1, index subroutine 2, material data, element number, function, node position vector dimension, material data vector dimension, dummy variable for output matrix

Functionality: Integrates function over square master element using a two-point Gaussian quadrature rule. Calculates entire 8x8 element matrix for elasticity stiffness or mass at once.

Returns: Matrix of element stiffness or mass values ready for assembly into global matrix.

Subroutine: GaussIntegrator4

Accepts: node position vector, index subroutine 1, index subroutine 2, material data, element number, function, node position vector dimension, material data vector dimension, dummy variable for output vector

Functionality: Integrates function over square master element using a two-point Gaussian quadrature rule. Calculates entire 8x1 element vector for elasticity load.

Returns: Vector of element load values ready for assembly into global vector.

Subroutine: elaskelem

Accepts: node position vector, node position vector dimension, material data vector, element number, dummy variable for output matrix

Functionality: Calls integrator for elasticity stiffness to produce element stiffness matrix.

Returns: Matrix of element values ready for assembly into global matrix.

Subroutine: elasmelem

Accepts: node position vector, node position vector dimension, material data vector, element number, dummy variable for output matrix

Functionality: Calls integrator for elasticity mass to produce element mass matrix.

Returns: Matrix of element values ready for assembly into global matrix.

Subroutine: elasfelem

Accepts: node position vector, node position vector dimension, material data vector 1, material data vector 2, element number, dummy variable for output vector

Functionality: Calls integrator for elasticity load to produce element load vector.

Returns: Vector of element values ready for assembly into global vector.

Subroutine: elas****elem

Accepts: node position vector, node position vector dimension, material data vector, element number, dummy variable for output vector

Functionality: Calls line integrator to produce boundary contributions to local load vector. Depending on the value of ****, it integrates the top, left, right or bottom of the element.

Returns: Vector of element values ready for assembly into global vector.

Subroutine: diffkelem

Accepts: node position vector, node position vector dimension, material data vector, element number, shape function 1, shape function 2, dummy variable for output scalar

Functionality: Calls integrator for diffusion stiffness to produce element stiffness matrix.

Returns: Scalar element value ready for assembly into global matrix.

Subroutine: diffmelem

Accepts: node position vector, node position vector dimension, material data vector, element number, shape function 1, shape function 2, dummy variable for output scalar

Functionality: Calls integrator for diffusion mass to produce element mass matrix.

Returns: Scalar element value ready for assembly into global matrix.

Subroutine: difffelem

Accepts: node position vector, node position vector dimension, material data vector, element number, shape function, dummy variable for output vector

Functionality: Calls integrator for diffusion load to produce element load vector.

Returns: Scalar element value ready for assembly into global vector.

Subroutine: ElemVolume

Accepts: node position vector, node position vector dimension, material data vector, element number, dummy variable for output scalar

Functionality: Calls volume integrator to solve for element volume. Does this exclusively on the diffusion mesh.

Returns: Volume of element.

Subroutine: MatrixAssemble

Accepts: function, node position vector, degree of freedom function, size of degree of freedom vector, function data, number of elements, dimension of node position vector, number of degrees of freedom per element, dummy variable for output matrix

Functionality: Creates local vector of degrees of freedom of size Indexbound for elements, loops through elements and assembles element contributions into global stiffness matrix. Function in this routine generates one entry at a time (i.e., used for diffusion stiffness and mass assembly). Note that only the lower triangular entries are calculated; the upper entries are acquired via symmetry.

Returns: Assembled global matrix.

Subroutine: StiffnessMatrixAssemble

Accepts: function, node position vector, degree of freedom function, size of degree of freedom vector, function data, number of elements, dimension of node position vector, number of degrees of freedom per element, dummy variable for output matrix

Functionality: Creates local vector of degrees of freedom of size Indexbound for elements, loops through elements and assembles element contributions into global stiffness matrix. Function in this routine generates entire element matrix at one time (i.e., used for elasticity stiffness and mass assembly). Note that only the lower triangular entries are calculated; the upper entries are acquired via symmetry.

Returns: Assembled global matrix.

Subroutine: VectorAssemble

Accepts: function, node position vector, degree of freedom function, size of degree of freedom vector, function data 1, function data 2, number of elements, dimension of node position vector, number of degrees of freedom per element, dummy variable for output vector

Functionality: Creates local vector of degrees of freedom of size Indexbound for elements, loops through elements and assembles element contributions into global load vector. Function in this routine generates entire element vector at one time (i.e., used for elasticity assembly).

Returns: Assembled global vector.

Subroutine: VectorAssemble2

Accepts: function, node position vector, degree of freedom function, size of degree of freedom vector, function data, number of elements, dimension of node position vector, number of degrees of freedom per element, dummy variable for output vector

Functionality: Creates local vector of degrees of freedom of size Indexbound for elements, loops through elements and assembles element contributions into global load vector. Function in this routine generates one element entry at a time (i.e., used for diffusion assembly).

Returns: Assembled global vector.

Subroutine: BoundaryAssemble****

Accepts: function, node position vector, degree of freedom function, size of degree of freedom vector, function data 1, function data 2, number of elements, dimension of node position vector, number of degrees of freedom per element, dummy variable for output vector

Functionality: Creates local vector of degrees of freedom of size Indexbound for elements, loops through elements and assembles element contributions into global load vector. Function in this routine generates entire element vector at one time (i.e., used for elasticity assembly). The BoundaryAssemble routines handle the boundary condition contribution on an element-by-element basis.

Returns: Assembled global vector.

Subroutine: DispFinalize

Accepts: vector of nodal values, number of degrees of freedom, vector of Dirichlet nodes, number of Dirichlet nodes, dummy variable for output vector

Functionality: Accepts of vector of nodal values at non-Dirichlet points, generates a new vector large enough to accommodate these values plus the Dirichlet values, then assembles the vector of nodal values and the already-defined Dirichlet values into a finalized displacement (or flux) vector.
Returns: Finalized vector of nodal values for the primary field variable.

Subroutine: MatrixLump
Accepts: Input matrix, matrix dimension, dummy argument for output matrix
Functionality: Adds matrix values in each row together, and sets the diagonal entry for that row equal to the sum. Used to row-lump mass matrices.
Returns: Lumped matrix

Subroutine: DirichletApply
Accepts: Input matrix, vector of Dirichlet nodes, number of degrees of freedom, number of Dirichlet degrees of freedom, dummy variable for output matrix
Functionality: Loops through matrix entries, sets to zero if row or column number is present in the Dirichlet vector. Useful only for zero-value Dirichlet conditions (all Dirichlet conditions handled in *TDMain* are zero-valued).
Returns: Matrix ready for conditioning subroutine.

Subroutine: VectorConditioner
Accepts: Input vector, vector of Dirichlet nodes, number of degrees of freedom, number of Dirichlet degrees of freedom, dummy variable for output vector

Functionality: Creates a new vector of size Input Vector minus the number of Dirichlet degrees of freedom, and populates it with the values in Input Vector not at Dirichlet nodes.

Returns: Load vector with Dirichlet conditions applied.

Subroutine: MatrixConditioner

Accepts: Matrix from DirichletApply routine, number of degrees of freedom, number of Dirichlet degrees of freedom, dummy variable for output matrix

Functionality: Accepts a matrix already operated upon by DirichletApply and does a zero-sum check for each row and column. If the sum of the square of all entries in a row or column is zero, it is removed from the matrix and the rest of the matrix entries are shifted left or up, depending on which part of the subroutine is being executed.

Returns: Stiffness or mass matrix with zero-valued Dirichlet conditions applied.

References

K. J. BATHE, Finite Element Procedures, Prentice Hall, Upper Saddle River, NJ, 1996.

E. B. BECKER, G. F. CAREY, and J. T. ODEN, Finite Elements: An Introduction, The University of Texas at Austin, TX, 1981.

M. L. BLEIBERG, L. J. JONES, and B. LUSTMAN, "Phase Changes in Pile-Irradiated Uranium-Base Alloys," Journal of Applied Physics, Vol. 27, pp. 1270-1283, November 1956.

I. F. BRIESMEISTER, Ed., "MCNP -A General Monte Carlo N-Particle Transport Code, Version 4C," LA-13709-M, Los Alamos National Laboratory, April 2000.

D. BURGREEN, "Thermoelastic Dynamics of Rods, Thin Shells and Solid Spheres", Nuclear Science and Engineering, Vol. 12, 203-217, 1962.

K. B. CARVER and T. G. TAXELIUS, "Fast Burst Reactor Facility Final Safeguards Report," KN-685-63-1 (FSR), Kaman Nuclear, Colorado Springs, CO, December 1963.

S. J. CHAPMAN, Fortran 90/95 for Scientists and Engineers, McGraw-Hill, New York, NY, 2004.

R. W. DICKINSON, *et al.*, "Safety Analysis Report for Army Pulse Radiation Facility Fast Pulse Reactor," BRL R1356, Aberdeen: Ballistic Research Laboratory, Aberdeen, MD, February 1967.

J. T. FORD, R.L COATS, J.J DAHL, S. A. WALKER, "Safety Analysis Report for the Sandia Pulsed Reactor Facility," SAND2003-2805, Sandia National Laboratories, Albuquerque, NM, December 2003.

Y. C. FUNG and P. TONG, Classical and Computational Solid Mechanics, World Scientific Publishing Co., River Edge, NJ, 2001.

K. F. GRAFF, Wave Motion in Elastic Solids, Dover Publications, New York, 1975.

G.E HANSEN, "Burst Characteristics Associated with the Slow Assembly of Fissionable Materials," LA-1441, Los Alamos National Laboratory, July 1952.

J. S. HENDRICKS, "MCNP4C," LANL Memo X-5: JSH-2000-30 (U), Los Alamos National Laboratory, February 2000.

D. L. HETRICK, Dynamics of Nuclear Reactors, University of Chicago Press, Chicago, 1971; reprinted by the American Nuclear Society, La Grange Park, IL, 1993.

K. G. HOGE, "Some Mechanical Properties of Uranium 10 wt-Percent Molybdenum Under Dynamic Tension Loads," UCRL-12357, Livermore: University of California Radiation Laboratory, Livermore, CA, February 1965.

T. J. R. HUGHES. The Finite Element Method: Linear Static and Dynamic Finite Element Analysis, Dover Publications, Mineola, NY, 2000.

KONOBEEVSKY, PRAVDYUK, and KUTAITSEV, "Proceedings of the International Conference on the Peaceful Uses of Atomic Energy," 7, pp. 433-440, 1956.

J. LAMARSH and A. BARATTA. Introduction to Nuclear Engineering 3rd Ed.. Prentice Hall Publishing, Upper Saddle, NJ, 2001.

J. P. LEHMAN, *et al.*, "Preliminary Safeguards Report for the Army Pulsed Reactor Assembly," AGN AN-1210, Aerojet-General Nucleonics, San Ramon, CA, 1964.

M. I. LUNDIN, "Health Physics Research Reactor Hazards Summary," ORNL-3248, Oak Ridge National Laboratory, Oak Ridge, TN, 1962.

J. D. MILLER, "Thermomechanical Analysis Of Fast-Burst Reactors," Proceedings of the Topical Meeting on Physics, Safety, and Applications of Pulse Reactors, Washington D.C., November 13–17, 1994.

D. M. MINNEMA, T. R. SCHMIDT, J. S. PHILBIN, J. W. BRYSON, V. SHAH, G. HOFFMAN, and R. E. ANDERSON, "The Final Report of the SPR III Fuel Technical Working Group (DRAFT)," National Nuclear Security Administration, Office of Defense Programs, U.S. Department of Energy, Washington, D.C., 2001.

R. B. OSWALD, *et al.*, "One-Dimensional Thermoelastic Response of Solids to Pulsed Energy Deposition," Journal of Applied Physics, Vol. 42, No. 9, pp. 3463-3473, 1971.

K. O. OTT and R. J. NEUHOLD, Introductory Nuclear Reactor Dynamics, American Nuclear Society, La Grange Park, IL,1985.

C. A. W. PETERSON and W. J. STEELE, "Study of the Effect of the Alloying on the Gamma-Phase Stability of Uranium Using Vacuum Differential Thermal Analysis," UCRL-7595, Livermore: University of California Radiation Laboratory, Livermore, CA, November 1963.

J. N. REDDY, An Introduction to the Finite Element Method, Third Edition, McGraw Hill, New York, NY, 2006.

J. A. REUSCHER, "Thermomechanical Analysis of Fast Burst Reactors," Proc. of the National Topical Meeting on Fast-Burst Reactors, part of AEC Symposium Proc., CONF-690102, January 1969.

J. A. REUSCHER, "Coupled Kinetic-Elasticity Calculations of Pulsed Reactor Performance," Proceedings of the Symposium, Dynamics of Nuclear Systems, Tucson: University of Arizona Press, University of Arizona, Tucson, AZ, 1970.

J. A. REUSCHER, "Analysis of Internal Heating Shock Effects in Reactor Fuel Components," Proceedings (Journal) of the First International Conference on Structural Mechanics in Reactor Technology, Berlin, Germany, Nuclear Engineering and Design, 18, No. 2, September 24, 1971.

J. A. REUSCHER, "In-Pile Testing of Fast-Burst Reactor Fuel Materials," AMC Research Reactor Symposium, White Sands Missile Range, New Mexico, February 20-21, 1973.

J.A REUSCHER and T.R SCHMIDT, "Overview of Sandia National Laboratories Pulse Nuclear Reactors," SAND94-2466C, Sandia National Laboratories, Albuquerque, NM, 1994.

R. J. VAN THYNE and D. J. MCPHERSON, "Transformation Kinetics of Uranium-Molybdenum Alloys," Trans. Am. Soc. Metals, Vol. 49, 1957.

S. C. WILSON, "Analysis of Dynamic Fuel Expansion Effects in a Fast Burst Reactor," University of Texas at Austin, TX, 2004.

T. F. WIMETT, *et al.*, "Godiva II - An Unmoderated Pulse-Irradiation Reactor," Nuclear Science and Engineering, Vol. 8, pp. 691-708, 1960.

T. F. WIMETT, "Dynamics and Power Prediction in Fission Bursts," Nuclear Science and Engineering, Vol. 110, pp. 209-236, 1992.

Z. ZUDANS, TSI CHU YEN and W. H. STEIGELMANN, Thermal Stress Techniques in the Nuclear Industry, American Elsevier Publishing Company, New York, 1965.

W. D. WILKINSON, Uranium Metallurgy, Volume 2: Uranium Corrosion and Alloys, Interscience Publishers, New York, pp. 1184-1190, 1962.

O. C. ZIENKIEWICZ, R. L. TAYLOR and J. Z. ZHU, The Finite Element Method: Its Basis and Fundamentals, Sixth Edition, Butterworth-Heinemann, Burlington, MA, 2006.

Vita

Stephen Wilson was born in West Islip, New York, in 1979, to parents Stephen and Laura Wilson. He attended high school in San Antonio, Texas, and obtained his Bachelor of Science degree in physics from Washington College in Maryland in 2003. He was accepted to the nuclear and radiation engineering program in the mechanical engineering department at the University of Texas in 2003, and obtained his Master of Science in Engineering degree in 2004. As of August 2006, he has completed his doctoral degree in the same discipline at the University of Texas at Austin.

Permanent address: 200 Gardenview,
San Antonio, TX 78213

This dissertation was typed by Stephen Wilson.