# Jacobian-Free Newton-Krylov Methods for Solving Nonlinear Neutronics/Thermal Hydraulic Equations

2.29 Numerical Fluid Mechanics Term Project

Bryan Herman

14 December 2011

# Contents

## 1 Objectives and Motivation

The main goal of this work is to learn how to implement the Jacobian-Free Newton Krylov framework. The following objectives are met in this work:

1. Learn how to solve and implement nonlinear equations with Newton's method,
2. Learn how the GMRES Krylov Subspace method works and implement it in MATLAB,
3. Compare solving an eigenvalue problem with conventional power iteration vs. JFNK,
4. Implement a coupled Neutronic/Thermal Hydraulic model to solve with JFNK both for steady state and transients,
5. Implement a fully implicit backward Euler approach for time discretization,
6. Combine models and solvers into a JFNK framework.

The JFNK method for solving coupled neutronics and thermal hydraulics equations is not commonly performed in the nuclear community. This method may be used in the coupling of neutronics and thermal hydraulics for coarse mesh finite difference acceleration of Monte Carlo neutron transport codes. This project is therefore an introduction to this method. A simple neutronic/thermal hydraulic model is used to test its implementation.

## 2 Introduction

A complete design of a nuclear reactor system is complex and involves many coupled physics. When performing reactor physics calculations, the design engineer is primarily concerned with the reactivity of the core, spatial power distribution and isotopics. This power distribution is usually used as an input condition for thermal hydraulic calculations which ensures that the core is cooled adequately. Whether performing reactor physics calculations for fuel management or for reactor safety, the distribution of neutrons (hence fission rate and power) and coolant density are highly coupled. The temperature of the fuel and coolant density distributions affect the probability of certain nuclear reactions such as fission.

Once example of a reactor safety calculation is the sudden ejection of a control rod. A common approach to solve this problem is to use operator or physics splitting. In this approach, a temperature/density distribution is assumed, a power distribution is calculated from neutronics and is fed to the thermal hydraulic equations to get a new temperature/density distribution. This iteration between operators continues until a steady state solution is found. A widely used core simulator that performs these transient calculations is the U.S. N.R.C. code PARCS [2] which is coupled to an external thermal hydraulic system code such as RELAP [13]. This iteration setup is shown in Fig. 2.1.

After a steady state coupled solution is found, the time-dependent solution can be calculated. In PARCS, this is done by an operator-split, explicit scheme as shown in Fig. 2.2. In their time-marching algorithm, the neutronics and thermal hydraulics are not converged in a given time step. Although this is an approximation, it has shown to be effective.

In this paper, a fully coupled time-dependent solution between neutronics and thermal hydraulics is performed. In this method, no operator splitting is performed and the equations are solved non-linearly at the same time. For this nonlinear system of equations, a Jacobian-Free Newton Krylov (JFNK) method is applied.

## 3 Model and Governing Equations

In this section, the reactor model will be discussed along with the governing equations that will be used to solve for physics. For the geometry, a one-dimensional slab reactor is assumed. A diagram of this model is shown in Fig. 3.1. In this geometry, the slab reactor (similar to modeling the axial direction of a fuel pin) is assumed to be 370 cm. To remove the heat produced from fission reactors in the core, a fluid is passed over the slab. Here, only one dimensional flow is considered and only the energy equation is applied since

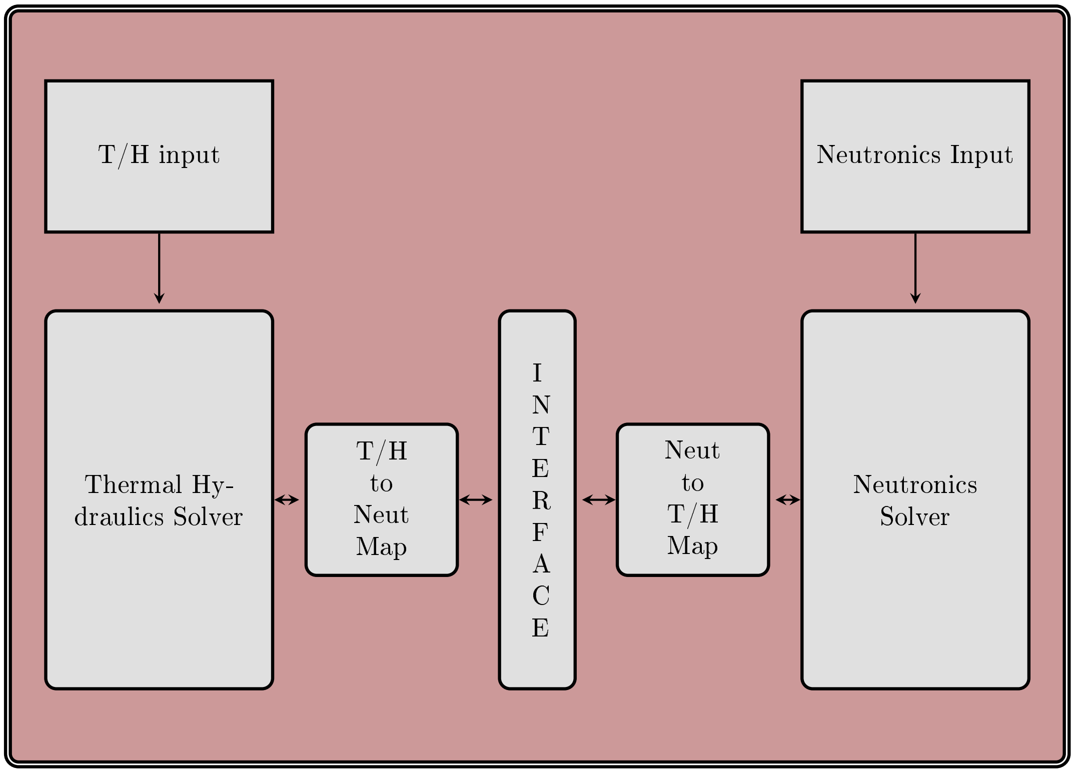


Figure 2.1. PARCS Coupling Structure

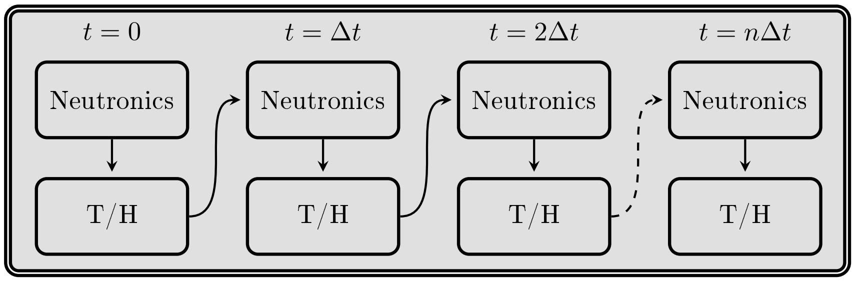


Figure 2.2. PARCS Marching Scheme

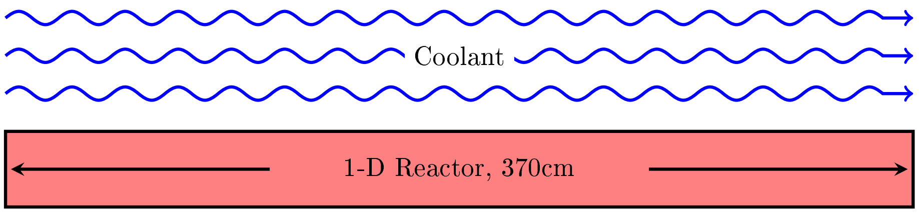


Figure 3.1. Geometry of Reactor Slab

the flow rate will be specified and remains constant. In addition, it will be assumed that the heat from the reactor is completely dumped into the coolant and there is no time constant for this process. Therefore, conduction and convective heat transfer equations are not applied. With these assumptions, the system of equations that remains is the simplest for these coupled physics. This is important since the JFNK nonlinear algorithm is of interest in this work.

## 3.1 Neutron Diffusion Equation

In this section, the governing equations to model the neutronics will be presented. The neutron transport equation is the most detailed form to describe how neutrons travel and interact in a medium. In time-dependent form, the neutron transport equation [5] is

$$\underbrace{\frac{1}{v}\frac{\partial\varphi}{\partial t}}_{\text{time}-\text{dependent}} + \underbrace{\mathbf{\Omega}\cdot\nabla\varphi\left(\mathbf{r},E,\mathbf{\Omega},t\right)}_{\text{neutron leakage}} + \underbrace{\Sigma_t\left(\mathbf{r},E,t\right)\varphi\left(\mathbf{r},E,\mathbf{\Omega},t\right)}_{\text{interation of neutrons with medium}} = \underbrace{Q\left(\mathbf{r},E,\mathbf{\Omega},t\right)}_{\text{neutron source}}, \tag{3.1}$$

$$\begin{aligned} Q\left(\mathbf{r},E,\mathbf{\Omega},t\right) &= \underbrace{\int_{4\pi} d^2\Omega' \int_0^\infty dE'\Sigma_s\left(\mathbf{r},E'\to E,\mathbf{\Omega}'\to\mathbf{\Omega},t\right)\varphi\left(\mathbf{r},E',\mathbf{\Omega}',t\right)}_{\text{neutrons scattering into phase space}} \\ &+ \underbrace{\frac{1}{4\pi}\frac{(1-\beta)}{k_{eff}}\int_0^\infty dE'\nu\Sigma_f\left(\mathbf{r},E'\to Et\right)\varphi\left(\mathbf{r},E',\mathbf{\Omega}',t\right)}_{\text{neutrons from prompt fissions}} + \underbrace{\frac{1}{4\pi}\sum_l \lambda_{d,l}c_l\left(\mathbf{r},t\right)}_{\text{neutrons from precursor decay}} . \end{aligned} \tag{3.2}$$

The variables in these formulas are as follows:

- $v$ - neutron speed
- $\varphi$ - angular neutron flux
- $t$ - time
- $\mathbf{\Omega}$ - unit vector of neutron travel
- $\mathbf{r}$ - spatial location of neutron
- $E$ - neutron energy
- $\Sigma_t$ - total macroscopic cross section
- $Q$ - neutron source
- $\Sigma_s$ - scattering macroscopic cross section
- $\beta$ - total delayed neutron fraction
- $\nu\Sigma_f$ - neutron fission production macroscopic cross section
- $\lambda_{d,l}$ - delayed neutron precursor decay constant with subscript $l$ for precursor group number
- $c_l$ - delayed neutron precursor concentration
- $k_{eff}$ - core multiplication factor

An important concept in reactor physics is $k_{eff}$ or the core multiplication factor. In modeling of reactors, if the steady state calculation is not perfectly balanced, neutron destruction does not equal neutron production, and an eigenvalue is introduced to state how far away from "steady" the reactor is. If $k_{eff}$ is unity then the neutron population is perfectly balanced. Therefore, when performing transient calculations, $k_{eff}$ from the steady state calculations must be used.

To model the concentration of precursors, the following equation is used for each precursor group $l$, which describes a balance of precursor production from fission reactions to their destruction from decay,

$$\frac{\partial c_l}{\partial t} = \underbrace{\frac{\beta}{k_{eff}} \int_0^\infty dE \int_0^\infty dE' \nu\Sigma_f \left(\mathbf{r}, E' \to Et\right) \varphi\left(\mathbf{r}, E', \mathbf{\Omega}', t\right) -}_{\text{neutrons from precursor decay}} \underbrace{\lambda_{d,l} c_l \left(\mathbf{r}, t\right)}_{\text{precursor decay}} . \tag{3.3}$$

The assumption is that the decay of one precursor isotope will give off one neutron. This is why the decay term shows up as a source of neutrons in the neutron balance equation. From these transport equations, the neutron diffusion equation can be derived. The form can be determined by expanding the angular flux and scattering source in Legendre polynomials and truncating after order 1. After the expansion and reduction, two equations result. The first represents neutron balance and the other is an equation that resembles Fick's Law (hence diffusion). They are

$$\frac{1}{v}\frac{\partial \phi}{\partial t} + \nabla \cdot \mathbf{J}\left(\mathbf{r}, E, t\right) + \Sigma_t \left(\mathbf{r}, E, t\right) \phi\left(\mathbf{r}, E, t\right) = \\ \int_0^\infty dE' \left[\Sigma_s \left(\mathbf{r}, E' \to E, t\right) + \frac{1-\beta}{k_{eff}} \nu\Sigma_f \left(\mathbf{r}, E' \to E, t\right)\right] \phi\left(\mathbf{r}, E', t\right) + \sum_l \lambda_{d,l} c_l \left(\mathbf{r}, t\right), \tag{3.4}$$

$$\mathbf{J}\left(\mathbf{r}, E, t\right) = -D\left(\mathbf{r}, E, t\right) \nabla\phi\left(\mathbf{r}, E, t\right). \tag{3.5}$$

The equation to model the precursor cursor concentration is

$$\frac{\partial c_l}{\partial t} = \frac{\beta}{k_{eff}} \int_0^\infty dE \int_0^\infty dE' \nu\Sigma_f \left(\mathbf{r}, E' \to E, t\right) \phi\left(\mathbf{r}, E', t\right) - \lambda_{d,l} c_l \left(\mathbf{r}, t\right). \tag{3.6}$$

The new variables introduced here are

- $\phi$ - scalar neutron flux, $\phi = \int_{4\pi} \varphi d^2\Omega$
- $\mathbf{J}$ - scalar neutron current, $\mathbf{J} = \int_{4\pi} \mathbf{\Omega}\varphi d^2\Omega$
- $D$ - neutron diffusion coefficient

### 3.1.1 One-dimensional One-group Transient Neutron Diffusion Equation

For this paper, only the one-dimensional one-energy group transient neutron diffusion equation is required. If Eq. (3.4) is integrated over all energies and only 1 precursor group is considered, the equations become

$$\frac{1}{v}\frac{\partial \phi}{\partial t} + \nabla \cdot \mathbf{J}\left(\mathbf{r}, t\right) + \Sigma_a \left(\mathbf{r}, t\right) \phi\left(\mathbf{r}, t\right) = \frac{1-\beta}{k_{eff}} \nu\Sigma_f \left(\mathbf{r}, t\right) \phi\left(\mathbf{r}, t\right) + \lambda_d c\left(\mathbf{r}, t\right), \tag{3.7}$$

$$\mathbf{J}\left(\mathbf{r}, t\right) = -D\left(\mathbf{r}, t\right) \nabla\phi\left(\mathbf{r}, t\right), \tag{3.8}$$

$$\frac{\partial c}{\partial t} = \frac{\beta}{k_{eff}} \nu\Sigma_f \left(\mathbf{r}, t\right) \phi\left(\mathbf{r}, t\right) - \lambda_d c\left(\mathbf{r}, t\right). \tag{3.9}$$

In Eq. (3.7), the neutron absorption macroscopic cross section is introduced. This cross section is defined as the difference between the total and scattering macroscopic cross section, $\Sigma_a \equiv \Sigma_t - \Sigma_s$. The equations can be further reduced to 1-dimension and become

$$\frac{1}{v}\frac{\partial \phi}{\partial t} + \frac{\partial J}{\partial x} + \Sigma_a \left(x, t\right) \phi\left(x, t\right) = \frac{1-\beta}{k_{eff}} \nu\Sigma_f \left(x, t\right) \phi\left(x, t\right) + \lambda_d c\left(x, t\right), \tag{3.10}$$

$$J\left(x, t\right) = -D\left(x, t\right) \frac{\partial \phi}{\partial x}, \tag{3.11}$$

$$\frac{\partial c}{\partial t} = \frac{\beta}{k_{eff}} \nu\Sigma_f \left(x, t\right) \phi\left(x, t\right) - \lambda_d c\left(x, t\right). \tag{3.12}$$

Equations (3.10), (3.11) and (3.12) form the set of transient neutronics equations that will be solved in this paper.

### 3.1.2 Steady State Form of the Neutron Diffusion Equation

Equations (3.10), (3.11) and (3.12) can be reduced to steady state form by removing the time-derivative term. This is shown as

$$\frac{dJ}{dx} + \Sigma_a (x) \phi (x) = \frac{1-\beta}{k_{eff}} \nu \Sigma_f (x) \phi (x) + \lambda_d c (x) , \tag{3.13}$$

$$J (x) = -D (x) \frac{d\phi}{dx}, \tag{3.14}$$

$$\lambda_d c (x) = \frac{\beta}{k_{eff}} \nu \Sigma_f (x) \phi (x) . \tag{3.15}$$

Equation (3.15) can be combined with Eq. (3.13) to give the final form of the 1-D neutron balance equation,

$$\frac{dJ}{dx} + \Sigma_a (x) \phi (x) = \frac{1}{k_{eff}} \nu \Sigma_f (x) \phi (x) . \tag{3.16}$$

Note that Eq. (3.16) is an eigenvalue problem since neutron current depends on flux through Fick's law. Here the spatial flux distribution represents the eigenfunction and $1/k_{eff}$ is the eigenvalue which is commonly denoted as $\lambda$ (not to be confused with the decay constant, $\lambda_d$).

## 3.2 Coupling Neutrons to Thermal Hydraulics

From the steady state neutronic analysis a neutron flux distribution is obtained. Since the flux is an eigenfunction, it must be normalized to some physical quantity to have meaning. In reactor physics the flux is normalized to reactor power. This step is known as flux-to-power normalization. This step can be calculated by computing the energy per fission multiplied by the fission reaction rate and integrated over all space and volume. This is represented mathematically as

$$Q_r = \tilde{c} \int_V d^3 r \int_0^\infty dE \kappa \Sigma_f (\mathbf{r}, E) \phi (\mathbf{r}, E) . \tag{3.17}$$

Here, the variables are

- $Q_r$ - reactor power
- $\tilde{c}$ - flux-to-power normalization constant
- $\kappa \Sigma_f$ - energy (from fission) deposition cross section

In one dimension and one energy group this is

$$Q_R = \tilde{c} \int_0^L dx \kappa \Sigma_f (x) \phi (x) . \tag{3.18}$$

Note that this step only needs to be performed during the steady state analysis since the transient analysis is not an eigenvalue problem. The transient equations are given the eigenvalue and eigenvector normalization constant, which are fixed throughout the transient.

Once this normalization constant is determined from the steady state calculation, the time-dependent spatial distribution of power density, $Q (x,t)$ can be calculated as

$$Q (x,t) = \tilde{c} \kappa \Sigma_f (x,t) \phi (x,t) . \tag{3.19}$$

This can be reduced to steady state to give

$$Q (x) = \tilde{c} \kappa \Sigma_f (x) \phi (x) . \tag{3.20}$$

## 3.3 Thermal Hydrualics - Energy Equation

The energy equation for a single phase fluid is given by [16]

$$\frac{\partial(\rho h)}{\partial t} + \nabla \cdot (\rho h \mathbf{u}) = -\nabla \cdot \mathbf{q}'' + q''' + \frac{Dp}{Dt} + \Phi. \tag{3.21}$$

The variables in this equation are

- $\rho$ - density of fluid
- $h$ - enthalpy of fluid
- t - time
- $\mathbf{u}$ - velocity vector
- $\mathbf{q}''$ - heat flux vector
- $q'''$ - volumetric heat source
- $p$ - pressure
- $\Phi$ - dissipation function

Assuming inviscid flow, the energy equation reduces to

$$\frac{\partial(\rho h)}{\partial t} + \nabla \cdot (\rho h \mathbf{u}) = -\nabla \cdot \mathbf{q}'' + q'''. \tag{3.22}$$

Assuming one-dimensional flow and no heat flux, the energy becomes

$$\frac{\partial(\rho h)}{\partial t} + \frac{\partial(\rho h u)}{\partial x} = q'''. \tag{3.23}$$

It will be assumed that the energy produced from fissions will be a volumetric heat source in the energy equation (note this is not common since fissions occur in fuel and not in coolant). Multiplying this form of the energy equation by the "area", $A$, of the flow, it can be rewritten as follows

$$A\frac{\partial(\rho h)}{\partial t} + w\frac{\partial h}{\partial x} = q'. \tag{3.24}$$

The new variables that appear in the above equation are

- $w$ - mass flow rate calculated as $w = \rho u A$ which from the continuity equation must be constant spatially and therefore taken out of the differential
- $q'$ - linear heat rate calculated as $q' = q'''A$

If the variation of density from thermal expansion is neglected as a function of time the equation reduces to

$$\rho A\frac{\partial h}{\partial t} + w\frac{\partial h}{\partial x} = q'. \tag{3.25}$$

Using the constitutive relation between enthalpy and temperature for an incompressible fluid that $dh = c_p dT$, the energy equation becomes

$$\rho A c_p\frac{\partial T}{\partial t} + w c_p\frac{\partial T}{\partial x} = q'. \tag{3.26}$$

This is the transient form of the energy equation that is solved in this paper. For steady state, the energy equation becomes

$$w c_p\frac{dT}{dx} = q'. \tag{3.27}$$

Note that if the energy equation is integrated over the whole reactor, the outlet temperature can be calculated with

$$T_{out} = T_{in} + \frac{Q_r}{w c_p}. \tag{3.28}$$

## 3.4 Thermal Hydraulic to Neutronic Coupling

The macroscopic cross sections and diffusion coefficient that were described in the neutronics formulation depend on density of the coolant. This due to the fact that water has dual purposes, as it functions as a coolant and slows down neutrons. Slow neutrons have a higher probability of causing a fission reaction. This process is almost analogous to billiard ball elastic collisions. Thus, if the density of water is low, it is less effective at slowing down neutrons and will have a negative effect on the rate at which fissions occur. This is why nuclear reactors are somewhat self-regulating. If the power suddenly increases in a region of the core, the temperature will increase lowering the density and thus decreasing the fission rate and power. Note that this may not be the dominant effect as there are other nuclear processes that will also have a negative effect on the fission rate such as fuel temperature. In this paper, only the effect of density is modeled.

Once the temperature distribution is known, the density distribution can be calculated from the equation of state for water at the specified system pressure. Thus,

$$\rho\left(x,t\right) = \rho\left(T\left(x,t\right),p\right). \tag{3.29}$$

The state equation is evaluated in the X-Steam look-up tables for MATLAB [6]. Since the macroscopic cross sections and diffusion coefficient are, to first order, linear with density. The following formulas can be used to describe the dependence of these diffusion theory parameters on density:

$$\Sigma_a\left(x,t\right) = \Sigma_a^{ref} + \frac{\partial \Sigma_a}{\partial \rho}\left[\rho\left(x,t\right) - \rho^{ref}\right], \tag{3.30}$$

$$\nu\Sigma_f\left(x,t\right) = \nu\Sigma_f^{ref} + \frac{\partial \nu\Sigma_f}{\partial \rho}\left[\rho\left(x,t\right) - \rho^{ref}\right], \tag{3.31}$$

$$D\left(x,t\right) = D^{ref} + \frac{\partial D}{\partial \rho}\left[\rho\left(x,t\right) - \rho^{ref}\right], \tag{3.32}$$

$$\kappa\Sigma_f\left(x,t\right) = \kappa\Sigma_f^{ref} + \frac{\partial \kappa\Sigma_f}{\partial \rho}\left[\rho\left(x,t\right) - \rho^{ref}\right]. \tag{3.33}$$

For each of the above equations, there is a reference value and are described in Section 9.

## 3.5 Calculation of Reference Neutronic Parameters

To calculate macroscopic cross sections and diffusion coefficients for core simulation, the 2-D neutron transport equation is solved. To solve this complicated PDE in space angle and energy, a stochastic Monte Carlo code, Serpent, is used to generate these parameters [10]. For this calculation a typical nuclear fuel rod is used depicted in Fig. 3.2.

For the Seabrook nuclear reactor [16], the inlet coolant temperature of the core is 293.1 °C and the outlet temperature is 326.8 °C. The core average temperature is taken as the straight average of these two values to give

$$T^{ref} = 310\,^{\circ}\mathrm{C}.$$

The pressure of the Seabrook nuclear reactor is 155 bar. Therefore using X-Steam, the density of water at this pressure and average temperature is

$$\rho^{ref} = 0.705\,\mathrm{g/cc}.$$

The mass flow rate and flow area are set as $w = 0.335\,\mathrm{kg/s}$ and $A = 0.879\,\mathrm{cm}^2$. The input file for the Serpent code with these reference conditions is listed in Appendix C. The reference neutronic parameters from the Serpent code (neglecting their associated uncertainties from the stochastic process) are as follows:

$$\Sigma_a^{ref} = 2.27516 \times 10^{-2}\,\mathrm{cm}^{-1},$$

$$\nu\Sigma_f^{ref} = 3.13791 \times 10^{-2}\,\mathrm{cm}^{-1},$$

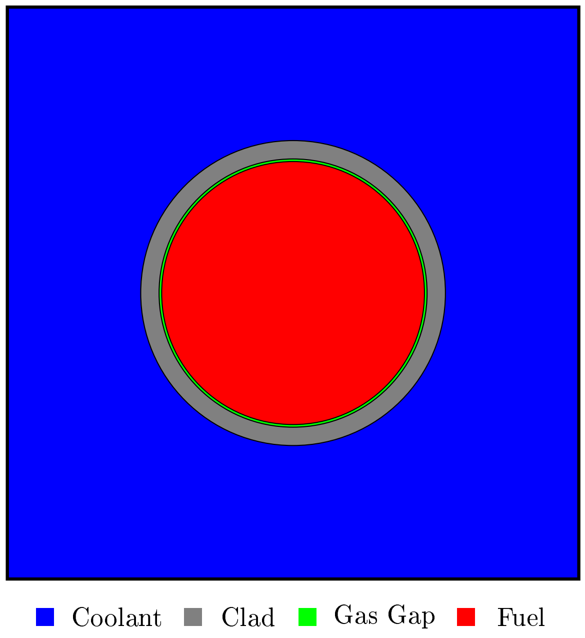


Figure 3.2. Typical 2-D Cross Section of a Fuel Rod unit cell

$$D^{ref} = 8.85342 \times 10^{-1}\,\text{cm},$$

$$\kappa\Sigma_f^{ref} = 4.13494 \times 10^{-13}\,\text{cm}^{-1}.$$

Perturbations of -10%, -5%, +5% and +10% were made to the reference density to obtain the dependence of the above parameters on density. These points are then fit with a linear regression to determine the slope of the data. This regression was performed for each of the parameters above, as shown in Figs. 3.3-3.6.

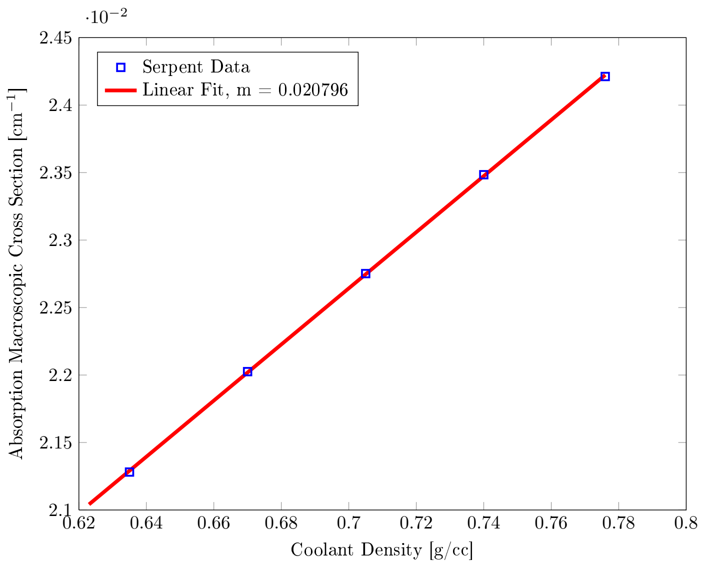


Figure 3.3. Dependence of Absorption Macroscopic Cross Section on Density

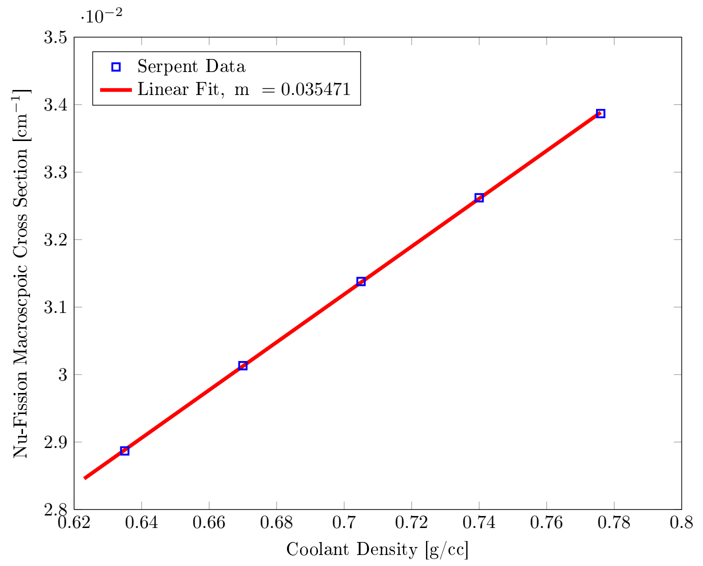


Figure 3.4. Dependence of Fission Neutron Production Macroscopic Cross Section on Density

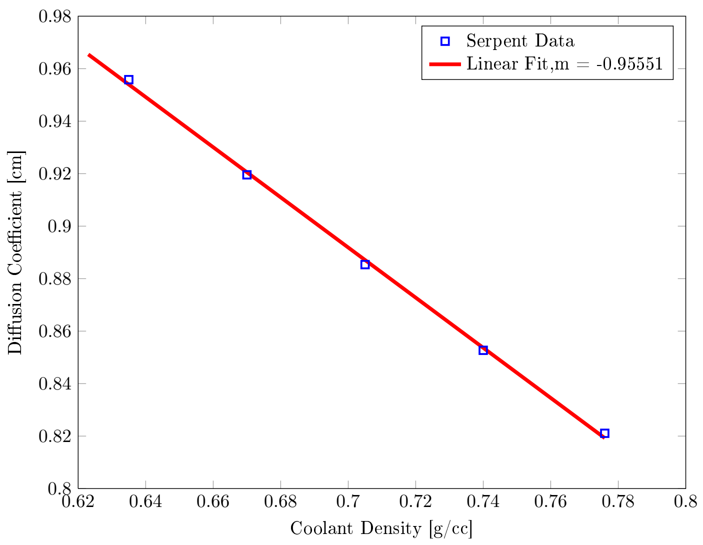


Figure 3.5. Dependence of Diffusion Coefficient on Density

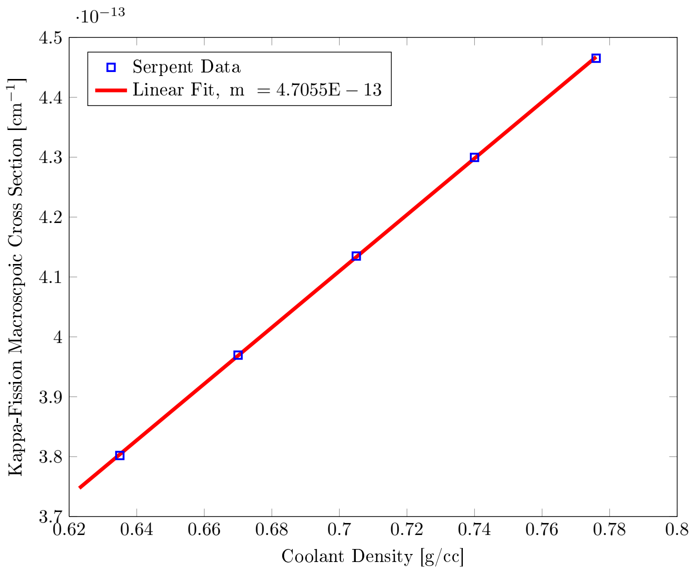


Figure 3.6. Dependence of Energy Deposition Macroscopic Cross Section on Density

From each of the figures, it can be observed that the trend of the data is linear. The slopes of the regressions are as follows:

$$\frac{\partial \Sigma_a}{\partial \rho} = 0.020796\,\mathrm{cm}^2/\mathrm{g},$$

$$\frac{\partial \nu\Sigma_f}{\partial \rho} = 0.035471\,\mathrm{cm}^2/\mathrm{g},$$

$$\frac{\partial D}{\partial \rho} = -0.95551\,\mathrm{cm}^4/\mathrm{g},$$

$$\frac{\partial \kappa\Sigma_f}{\partial \rho} = 4.7055 \times 10^{-13}\,\mathrm{cm}^2/\mathrm{g}.$$

From the results, the macroscopic cross sections have a positive slope while the diffusion coefficient has a negative slope. The cross section dependence makes sense because as density increases, neutrons slow down more effectively and increase the rate of fission. The opposite is true for the diffusion coefficient. If the density increases, neutrons are more effectively slowed and do not diffuse as far. Therefore, if the density goes down there will be a negative effect on producing more neutrons from fission as the cross section will go down and neutrons will diffuse further and have a higher probability of leaking out of the core.

# 4 Discretization of Equations

In this section, the governing 1-D equations presented in Section 3 are discretized in space and time. Before a transient calculation can be performed, a steady solution must be determined. Therefore, the discretization of the steady state form of the equations will be presented first. It is assumed that the grid is discretized uniformly over the slab.

## 4.1 Steady State Equations

For the discretization in space, a second order finite volume method will be used. This discretization is shown in Fig. 4.1.

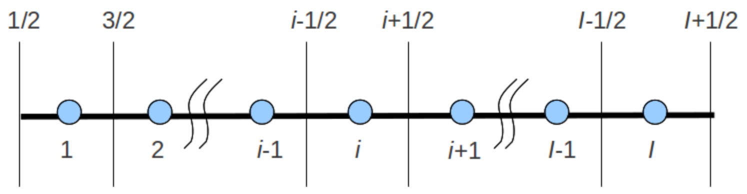


Figure 4.1. Spatial Discretization of 1-D Geometry

### 4.1.1 Neutronics

We can integrate each term in Eq. (3.16) over cell $i$,

$$\int_{x_{i-1/2}}^{x_{i+1/2}} dx \frac{dJ}{dx} + \int_{x_{i-1/2}}^{x_{i+1/2}} dx \Sigma_a (x) \phi (x) = \int_{x_{i-1/2}}^{x_{i+1/2}} dx \frac{1}{k_{eff}} \nu \Sigma_f (x) \phi (x) . \tag{4.1}$$

The average flux in cell $i$ can be calculated as

$$\bar{\phi}_i = \frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} dx \phi (x) . \tag{4.2}$$

The average flux for cell $i$ is then taken to be the value at the center of the cell,

$$\bar{\phi}_i = \bar{\phi}_i (x_i) = \phi (x_i) + O \left( \Delta x^2 \right) . \tag{4.3}$$

This is a second order approximation. It is also assumed in this derivation that neutronic parameters are spatially constant in a cell. Performing the integration over cell $i$, the neutron balance equation becomes

$$J_{i+1/2} - J_{i-1/2} + \Sigma_{a,i} \bar{\phi}_i \Delta x = \frac{1}{k_{eff}} \nu \Sigma_{f,i} \bar{\phi}_i \Delta x. \tag{4.4}$$

The fluxes that show up in finite volume equations are actually the neutron current, NOT the neutron flux. From Fick's Law in Eq. (3.14), the fluxes at the surfaces of the mesh cell are given by

$$J_{i+1/2} = -D_i \left. \frac{d\phi}{dx} \right|_{i+1/2} \qquad J_{i-1/2} = -D_i \left. \frac{d\phi}{dx} \right|_{i-1/2} . \tag{4.5}$$

A second order central difference scheme is applied to approximate the derivative of the flux at the boundary. The diffusion coefficients in adjacent cells do not necessarily need to be the same, however, the neutron current at the interface must be equivalent. For the $i + 1/2$ interface, the current from the left cell is equated to the current from the right cell,

$$-D_i \left. \frac{d\phi}{dx} \right|_{i+1/2} = -D_{i+1} \left. \frac{d\phi}{dx} \right|_{i+1/2} . \tag{4.6}$$

Applying a second order finite difference to the derivatives, the current continuity becomes

$$-D_i \frac{\phi_{i+1/2} - \bar{\phi}_i}{\Delta x/2} = -D_{i+1} \frac{\bar{\phi}_{i+1} - \phi_{i+1/2}}{\Delta x/2}. \tag{4.7}$$

The flux at the interface can be represented as

$$\phi_{i+1/2} = \frac{2}{\Delta x} \frac{D_{i+1}}{D_{i+1} + D_i} \left( \bar{\phi}_{i+1} - \bar{\phi}_i \right) . \tag{4.8}$$

Therefore the current or finite volume flux at the right interface is

$$J_{i+1/2} = -\frac{2}{\Delta x} \frac{D_{i+1} D_i}{D_{i+1} + D_i} \left( \bar{\phi}_{i+1} - \bar{\phi}_i \right) . \tag{4.9}$$

Notice that if the diffusion coefficient is equivalent in adjacent cells, the equation reduces to the simple second order central difference. Similarly for the left interface, the current is

$$J_{i-1/2} = -\frac{2}{\Delta x}\frac{D_i D_{i-1}}{D_i + D_{i-1}}\left(\bar{\phi}_i - \bar{\phi}_{i-1}\right). \tag{4.10}$$

Substituting these currents into Eq. (4.4) the discretized diffusion equation is

$$-\frac{2}{\Delta x}\frac{D_{i+1}D_i}{D_{i+1}+D_i}\left(\bar{\phi}_{i+1}-\bar{\phi}_i\right) + \frac{2}{\Delta x}\frac{D_i D_{i-1}}{D_i + D_{i-1}}\left(\bar{\phi}_i - \bar{\phi}_{i-1}\right) + \Sigma_{a,i}\bar{\phi}_i\Delta x = \frac{1}{k_{eff}}\nu\Sigma_{f,i}\bar{\phi}_i\Delta x. \tag{4.11}$$

Grouping like terms on the left hand side of the equation and dividing by the cell volume,

$$\begin{aligned} -\frac{2}{\Delta x^2}\frac{D_i D_{i-1}}{D_i + D_{i-1}}\bar{\phi}_{i-1} + \left(\frac{2}{\Delta x^2}\frac{D_{i+1}D_i}{D_{i+1}+D_i} + \frac{2}{\Delta x^2}\frac{D_i D_{i-1}}{D_i + D_{i-1}} + \Sigma_{a,i}\right)\bar{\phi}_i - \frac{2}{\Delta x^2}\frac{D_{i+1}D_i}{D_{i+1}+D_i}\bar{\phi}_{i+1} \\ = \frac{1}{k_{eff}}\nu\Sigma_{f,i}\bar{\phi}_i. \end{aligned} \tag{4.12}$$

Since the neutron diffusion equation is a second order differential equation, two boundary conditions must be specified. A physical boundary condition is to say that once a neutron leaves the reactor it will never come back. This may not always be true, especially depending on how the boundaries are defined. Another neutronic parameter called an albedo is defined as the ratio of incoming current of neutrons to outgoing current of neutrons at an interface,

$$\beta = \frac{J_{in}}{J_{out}}. \tag{4.13}$$

For the left boundary the current is

$$J_{1/2} = -D_1 \left.\frac{d\phi}{dx}\right|_{1/2}. \tag{4.14}$$

The current at an interface is actually the net neutron current at that surface. This net current can always be decomposed into a partial current going to the right and a partial current going to the left. The relation between these partial currents to the net current is represented as

$$J = J_{right} - J_{left}. \tag{4.15}$$

Current to the right is taken as positive since the positive $x$ direction is to the right. Notice that depending on the boundary, the partial current to the right may be the same as the incoming current or the outgoing current. For the left boundary, the net current is

$$J_{1/2} = J_{in} - J_{out} = -D_1 \left.\frac{d\phi}{dx}\right|_{1/2}. \tag{4.16}$$

From neutron transport theory, the partial incoming and outgoing currents can be represent in terms of the flux at the boundary (Marshak Boundary Conditions [1]),

$$J_{left} = \frac{1}{4}\phi_{1/2} - \frac{1}{2}J_{1/2} \qquad J_{right} = \frac{1}{4}\phi_{1/2} + \frac{1}{2}J_{1/2}. \tag{4.17}$$

Comparing Eqs. (4.15) and (4.16), the Marshak boundary conditions are

$$J_{out} = \frac{1}{4}\phi_{1/2} - \frac{1}{2}J_{1/2} \qquad J_{in} = \frac{1}{4}\phi_{1/2} + \frac{1}{2}J_{1/2}.$$

These partial current equations can be substituted into Eq. (4.13) and the net current can be determined to be

$$J_{1/2} = -\frac{1}{2}\frac{1-\beta}{1+\beta}\phi_{1/2}. \tag{4.18}$$

Substituting this expression into Eq. (4.14) and taking a first order finite difference of the spatial derivative, Fick's Law becomes

$$-\frac{1}{2}\frac{1-\beta}{1+\beta}\phi_{1/2} = -D_1\frac{\bar{\phi}_1 - \phi_{1/2}}{\Delta x/2}. \tag{4.19}$$

The surface flux at the left boundary is determined to be

$$\phi_{1/2} = \frac{4\left(1+\beta\right)D_1}{4D_1\left(1+\beta\right) + \Delta x\left(1-\beta\right)}\bar{\phi}_1. \tag{4.20}$$

Substituting this equation back into Eq. (4.14) after applying the first order finite difference, the net current at the boundary is

$$J_{1/2} = -\frac{2D_1\left(1-\beta\right)}{4D_1\left(1+\beta\right) + \Delta x\left(1-\beta\right)}\bar{\phi}_1. \tag{4.21}$$

Using the exact same process (except incoming/outgoing current definitions switch) the net current on the right boundary is

$$J_{I+1/2} = \frac{2D_I\left(1-\beta\right)}{4D_I\left(1+\beta\right) + \Delta x\left(1-\beta\right)}\bar{\phi}_I. \tag{4.22}$$

The final form of the discretized equation for the left boundary is

$$\left(\frac{2}{\Delta x^2}\frac{D_2 D_1}{D_2 + D_1} + \frac{2}{\Delta x}\frac{\left(1-\beta\right)D_1}{4D_1\left(1+\beta\right) + \Delta x\left(1-\beta\right)} + \Sigma_{a,1}\right)\bar{\phi}_1 - \frac{2}{\Delta x^2}\frac{D_2 D_1}{D_2 + D_1}\bar{\phi}_2 = \frac{1}{k_{eff}}\nu\Sigma_{f,1}\bar{\phi}_1. \tag{4.23}$$

For the right boundary it is

$$-\frac{2}{\Delta x^2}\frac{D_I D_{I-1}}{D_I + D_{I-1}}\bar{\phi}_{I-1} + \left(\frac{2D_I\left(1-\beta\right)}{4D_I\left(1+\beta\right) + \Delta x\left(1-\beta\right)} + \frac{2}{\Delta x^2}\frac{D_I D_{I-1}}{D_I + D_{I-1}} + \Sigma_{a,I}\right)\bar{\phi}_I = \frac{1}{k_{eff}}\nu\Sigma_{f,I}\bar{\phi}_I. \tag{4.24}$$

Equations (4.12), (4.23) and (4.24) can be represented in matrix notation as

$$\mathbb{M}\bar{\mathbf{\Phi}} = \lambda\mathbb{F}\bar{\mathbf{\Phi}}, \tag{4.25}$$

where

- $\mathbb{M}$ is the neutron destruction operator,
- $\bar{\mathbf{\Phi}}$ is a vector of cell average fluxes,
- $\lambda$ is the eigenvalue of the system, which is $1/k_{eff}$,
- $\mathbb{F}$ is the neutron production operator.

The neutron destruction operator has a tridiagonal form while the production operator is a diagonal of fission neutron production macroscopic cross sections. Since this eigenvalue problem will be formulated in a nonlinear sense, another equation is needed to constrain the eigenvector. The common approach is to make the L2-norm of the eigenvector be unity [4],

$$\left\|\bar{\mathbf{\Phi}}\right\|_2 = 1. \tag{4.26}$$

### 4.1.2 Coupling Neutrons to Thermal Hydraulics

The normalization condition in Eq. (3.18) can be converted into a summation of discrete volumes as

$$Q_R = \tilde{c}\int_0^L dx\kappa\Sigma_f\left(x\right)\phi\left(x\right) = \tilde{c}\sum_i \kappa\Sigma_{f,i}\bar{\phi}_i\Delta x = \tilde{c}\kappa\mathbf{\Sigma}_f^{\mathrm{T}}\bar{\mathbf{\Phi}}\Delta x. \tag{4.27}$$

Once the normalization constant is determined, the power in each volume can be determined by integrating the power density (see Eq. (3.19)) over a cell volume. The resulting formula is

$$Q_i = \int_{x_{i-1/2}}^{x_{i+1/2}} \tilde{c}\kappa\Sigma_f\phi\left(x\right) = \tilde{c}\kappa\Sigma_f\bar{\phi}_i\Delta x. \tag{4.28}$$

In matrix form, the above equation is represented by

$$\mathbf{Q} = \tilde{c}\mathbb{E}\bar{\mathbf{\Phi}}\Delta x, \tag{4.29}$$

where the energy deposition operator $\mathbb{E}$ is a diagonal matrix of the energy deposition cross section.

### 4.1.3 Energy Equation

The steady state energy can be integrated over cell $i$ to give

$$\int_{x_{i-1/2}}^{x_{i+1/2}} dx\frac{dT}{dx} = \int_{x_{i-1/2}}^{x_{i+1/2}} dx\frac{q'}{wc_p}. \tag{4.30}$$

After integration,

$$T_{i+1/2} - T_{i-1/2} = \frac{Q_i}{wc_p}. \tag{4.31}$$

We can define the average cell temperature and take it at the center of the cell

$$\bar{T}_i \approx \frac{1}{\Delta x}\int_{x_{i-1/2}}^{x_{i+1/2}} dxT\left(x\right). \tag{4.32}$$

The surface temperature on the left can be related to the temperature in the left adjacent cell as

$$T_{i-1/2} = \bar{T}_{i-1} + \frac{Q_{i-1}}{2wc_p} \tag{4.33}$$

since only half of the energy is deposited between the center of the cell and the surface. Similarly, the right surface temperature can be related to the cell average temperature as

$$T_{i+1/2} = \bar{T}_i + \frac{Q_i}{2wc_p}. \tag{4.34}$$

These two approximations are known as Upwind approximations. Substituting these surface temperatures into Eq. (4.31) gives a relation between cell average temperatures

$$\bar{T}_i - \bar{T}_{i-1} = \frac{1}{2wc_p}Q_{i-1} + \frac{1}{2wc_p}Q_i. \tag{4.35}$$

This can be represented in matrix notation as

$$\mathbb{S}\bar{\mathbf{T}} = \mathbb{R}\mathbf{Q}, \tag{4.36}$$

where $\mathbb{S}$ is the temperature operator that contains a diagonal and a subdiagonal and $\mathbb{R}$ is the energy operator that is comprised of a diagonal and subdiagonal.

### 4.1.4 Energy to Neutronic Coupling

These coupling equations can simply be converted for a whole cell. The cell average density can be related to the cell average density as

$$\rho_i = \rho\left(\bar{T}_i, p\right). \tag{4.37}$$

In vector form this is

$$\mathcal{P} = \rho\left(\bar{\mathbf{T}}, p\right). \tag{4.38}$$

The neutronic parameters can be related to this cell average density as

$$\Sigma_{a,i} = \Sigma_a^{ref} + \frac{\partial \Sigma_a}{\partial \rho}\left[\rho_i - \rho^{ref}\right], \tag{4.39}$$

$$\nu\Sigma_{f,i} = \nu\Sigma_f^{ref} + \frac{\partial \nu\Sigma_f}{\partial \rho}\left[\rho_i - \rho^{ref}\right], \tag{4.40}$$

$$D_i = D^{ref} + \frac{\partial D}{\partial \rho}\left[\rho_i - \rho^{ref}\right], \tag{4.41}$$

$$\kappa\Sigma_{f,i} = \kappa\Sigma_f^{ref} + \frac{\partial \kappa\Sigma_f}{\partial \rho}\left[\rho_i - \rho^{ref}\right]. \tag{4.42}$$

In vector form these are written as

$$\mathbf{\Sigma}_a = \Sigma_a^{ref} + \frac{\partial \Sigma_a}{\partial \rho}\left[\mathcal{P} - \rho^{ref}\right], \tag{4.43}$$

$$\nu\mathbf{\Sigma}_f = \nu\Sigma_f^{ref} + \frac{\partial \nu\Sigma_f}{\partial \rho}\left[\mathcal{P} - \rho^{ref}\right], \tag{4.44}$$

$$\mathbf{D} = D^{ref} + \frac{\partial D}{\partial \rho}\left[\mathcal{P} - \rho^{ref}\right], \tag{4.45}$$

$$\kappa\mathbf{\Sigma}_f = \kappa\Sigma_f^{ref} + \frac{\partial \kappa\Sigma_f}{\partial \rho}\left[\mathcal{P} - \rho^{ref}\right]. \tag{4.46}$$

## 4.2 Transient Equations

In this section, the time-dependent form of the discretized equations will be formulated. Since the spatial operators have been defined in the previous subsection, these will also be used in the time-dependent equations. Note that Eqs. (4.27), (4.29), (4.38), (4.43), (4.44), (4.45) and (4.46) do not have a time-dependent term.

### 4.2.1 Neutronics-Flux

For an interior computational cell, the discretized neutron diffusion equation in time-dependent (1-D, 1-energy) form is

$$\frac{1}{v}\frac{d\bar{\phi}_i}{dt} + -\frac{2}{\Delta x^2}\frac{D_i D_{i-1}}{D_i + D_{i-1}}\bar{\phi}_{i-1} + \left(\frac{2}{\Delta x^2}\frac{D_{i+1} D_i}{D_{i+1} + D_i} + \frac{2}{\Delta x^2}\frac{D_i D_{i-1}}{D_i + D_{i-1}} + \Sigma_{a,i}\right)\bar{\phi}_i - \frac{2}{\Delta x^2}\frac{D_{i+1} D_i}{D_{i+1} + D_i}\bar{\phi}_{i+1} = \frac{1-\beta}{k_{eff}}\nu\Sigma_{f,i}\bar{\phi}_i + \lambda_d \bar{c}_i. \tag{4.47}$$

Applying the spatial operators, this is reduced in matrix form to

$$\frac{1}{v}\frac{d\bar{\mathbf{\Phi}}}{dt} + \mathbb{M}\bar{\mathbf{\Phi}} = (1-\beta)\,\lambda\mathbb{F}\bar{\mathbf{\Phi}} + \lambda_d \bar{\mathbf{c}}. \tag{4.48}$$

Applying an implicit Euler time discretization scheme,

$$\frac{1}{v}\frac{\bar{\mathbf{\Phi}}^{n+1}-\bar{\mathbf{\Phi}}^{n}}{\Delta t}+\mathbb{M}\bar{\mathbf{\Phi}}^{n+1}=(1-\beta)\,\lambda\mathbb{F}\bar{\mathbf{\Phi}}^{n+1}+\lambda_d\bar{\mathbf{c}}^{n+1}, \tag{4.49}$$

where $n$ denotes the time step. In residual form,

$$\bar{\mathbf{\Phi}}^{n+1}-\bar{\mathbf{\Phi}}^{n}+v\Delta t\left(\mathbb{M}\bar{\mathbf{\Phi}}^{n+1}-(1-\beta)\,\lambda\mathbb{F}\bar{\mathbf{\Phi}}^{n+1}-\lambda_d\bar{\mathbf{c}}^{n+1}\right)=0. \tag{4.50}$$

### 4.2.2 Neutronics-Precursors

The precursor concentration from Eq. (3.12) can be cast in finite volume form as

$$\frac{d\bar{\mathbf{c}}}{dt}=\beta\lambda\mathbb{F}\bar{\mathbf{\Phi}}-\lambda_d\bar{\mathbf{c}}. \tag{4.51}$$

Discretizing the time derivative with implicit Euler,

$$\frac{\bar{\mathbf{c}}^{n+1}-\bar{\mathbf{c}}^{n}}{\Delta t}=\beta\lambda\mathbb{F}\bar{\mathbf{\Phi}}^{n+1}-\lambda_d\bar{\mathbf{c}}^{n+1}. \tag{4.52}$$

In residual form,

$$\bar{\mathbf{c}}^{n+1}-\bar{\mathbf{c}}^{n}+\Delta t\left(\lambda_d\bar{\mathbf{c}}^{n+1}-\beta\lambda\mathbb{F}\bar{\mathbf{\Phi}}^{n+1}\right)=0. \tag{4.53}$$

### 4.2.3 Energy Equation

The time-dependent form of the energy equation that will be used in this model is shown in Eq. 3.26. Rearranging the equation,

$$\frac{\rho A\Delta x}{w}\frac{d\bar{T}_i}{dt}+\bar{T}_i-\bar{T}_{i-1}=\frac{1}{2wc_p}Q_{i-1}+\frac{1}{2wc_p}Q_i. \tag{4.54}$$

Applying the spatial operators defined for the steady state equations,

$$\frac{\mathcal{P}A\Delta x}{w}\frac{d\bar{\mathbf{T}}}{dt}+\mathbb{S}\bar{\mathbf{T}}=\mathbb{R}\mathbf{Q}. \tag{4.55}$$

Using implicit Euler to discretize the time derivative,

$$\frac{\mathcal{P}^{n+1}A}{w}\frac{\bar{\mathbf{T}}^{n+1}-\bar{\mathbf{T}}^{n}}{\Delta t}+\mathbb{S}\bar{\mathbf{T}}^{n+1}=\mathbb{R}\mathbf{Q}^{n+1}. \tag{4.56}$$

In residual form,

$$\bar{\mathbf{T}}^{n+1}-\bar{\mathbf{T}}^{n}+\frac{w\Delta t}{\mathcal{P}^{n+1}A\Delta x}\left(\mathbb{S}\bar{\mathbf{T}}^{n+1}-\mathbb{R}\mathbf{Q}^{n+1}\right)=0. \tag{4.57}$$

Note that the density vector is in the denominator of the coefficient fraction.

# 5 Newton's Method

To solve the coupled neutronic/thermal hydraulic problem, a nonlinear method must be used. A common approach for solving nonlinear equations is to employ Newton's method [8]. The algorithm for Newton's method is presented in Algorithm 1.

**Algorithm 1** Newton's Method

```
1: for $n = 1, 2, 3, ...$ do
2:     evaluate residual, $\mathbf{F}(\mathbf{x}_n)$
3:     test for convergence
4:     evaluate Jacobian, $\mathbb{J}(\mathbf{x}_n)$
5:     solve $d\mathbf{x} = \mathbb{J}(\mathbf{x}_n)^{-1}\mathbf{F}(\mathbf{x}_n)$
6:     compute next guess, $\mathbf{x}_{n+1} = \mathbf{x}_n + d\mathbf{x}$
7: end for
```

Therefore, a set of residual equations must be formulated such that

$$\mathbf{F}(\mathbf{x}) = 0. \tag{5.1}$$

where $\mathbf{F}$ is the residual vector and $\mathbf{x}$ is the unknown vector. From these residual equations, a Jacobian matrix can be constructed by taking the partial derivative of each residual equation by the variables in the unknown vector. To test for convergence, the norm of the residual is usually compared to some tolerance. This termination criteria is shown as

$$\|\mathbf{F}(\mathbf{x})\|_2 < tol. \tag{5.2}$$

This nonlinear tolerance is user-defined and arbitrary. We only note that this value be compared to the tolerance used when solving the linear system equation with an iterative method (see step 5 in Algorithm 1).

In the coupled system of neutronic and thermal hydraulic equations, an analytic Jacobian cannot be determined since the state equation for water is a look-up table. One could fit an analytic curve to the state equation to describe the dependence of density on temperature. However, in this application, the Jacobian will be approximated. This process is discussed in Section 7. Therefore, direct methods cannot be utilized when solving this linear system of equations with Newton's method. Instead, an iterative GMRES Krylov subspace method is used and is described in detail in Section 6.

# 6 Krylov Subspace Methods

In Krylov methods, the goal is to solve $\mathbb{A}\mathbf{x} = \mathbf{b}$. Krylov methods fall into the category of iterative projection methods. In projection methods, an approximate solution to the vector $\mathbf{x}$, denoted as $\hat{\mathbf{x}}$, is determined from a projection of the system onto some subspace. In Krylov methods, a Krylov subspace, $\mathcal{K}_n$, has the form

$$\mathcal{K}_n(\mathbb{A}, \mathbf{v}) = \text{span}\left\{\mathbf{v}, \mathbb{A}\mathbf{v}, \mathbb{A}^2\mathbf{v}, ..., \mathbb{A}^{n-1}\mathbf{v}\right\}, \tag{6.1}$$

where $n$ is a dimension of the subspace which is $m \times n$, $\mathbb{A}$ is an $m \times m$ matrix and $\mathbf{v}$ is a vector of length $m$ [15]. Here, the vectors $\mathbf{v}$, $\mathbb{A}\mathbf{v}$... form a basis of $\mathcal{K}_m$. Arnoldi's method allows for a general non-Hermitian matrix to be orthogonally projected onto $\mathcal{K}_n$. According to Saad, this procedure was introduced as a means of reducing dense matrices into Hessenberg form. The power of the Arnoldi Iteration is that with a small number of steps to create a Hessenberg matrix, the eigenvalues of this matrix approximate the eigenvalues of the original matrix. This is very important and powerful for large sparse linear systems of equations.

## 6.1 Arnoldi Iteration

The Arnoldi process is a way to transform a matrix to Hessenberg form. Trefethen's notation will be used in defining the Arnold iteration [17]. This can be represented as

$$\mathbb{A}\mathbb{Q} = \mathbb{Q}\mathbb{H}. \tag{6.2}$$

Here, $\mathbb{A}$ is the coefficient matrix, $\mathbb{Q}$ is unitary and $\mathbb{H}$ is a matrix in Hessenberg form. A Hessenberg matrix that is $n \times n$ has the form,

$$\mathbb{H} = \begin{bmatrix} h_{11} & & \cdots & h_{1n} \\ h_{21} & h_{22} & & \\ & \ddots & \ddots & \vdots \\ & & h_{n,n-1} & h_{n,n} \end{bmatrix}. \tag{6.3}$$

Since matrix $\mathbb{A}$ may be very large, a full reduction to Hessenberg form may not be feasible. Rather, the first $n$ columns are considered so that $\mathbb{Q}_n$ is a $m \times n$ matrix which contains the first $n$ columns of $\mathbb{Q}$,

$$\mathbb{Q}_n = [\mathbf{q}_1, \mathbf{q}_2 ..., \mathbf{q}_n]. \tag{6.4}$$

To set up the iteration, Eq. (6.2) becomes

$$\mathbb{A}\mathbb{Q}_n = \mathbb{Q}_{n+1}\widetilde{\mathbb{H}}_n. \tag{6.5}$$

In Eq. (6.5) $\widetilde{\mathbb{H}}_n$ is a $(n+1) \times n$ upper-left section of $\mathbb{H}$ and also of Hessenberg form,

$$\widetilde{\mathbb{H}}_n = \begin{bmatrix} h_{11} & & \cdots & h_{1n} \\ h_{21} & h_{22} & & \\ & \ddots & \ddots & \vdots \\ & & h_{n,n-1} & h_{n,n} \\ & & & h_{n+1,n} \end{bmatrix}.$$

If $\mathbb{A}$ is applied to the $n$-th column of $\mathbb{Q}_n$ in Eq. (6.5), the following formula can be derived:

$$\mathbb{A}\mathbf{q}_n = h_{1n}\mathbf{q}_1 + \cdots + h_{n,n}\mathbf{q}_n + h_{n+1,n}\mathbf{q}_{n+1}. \tag{6.6}$$

Thus, the next column of $\mathbb{Q}$ can be determined with

$$\mathbf{v} = \mathbb{A}\mathbf{q}_n - (h_{1n}\mathbf{q}_1 + \cdots + h_{n,n}\mathbf{q}_n) \tag{6.7}$$

$$\mathbf{q}_{n+1} = \mathbf{v}/h_{n+1,n}, \tag{6.8}$$

where $\mathbf{v}$ is just a temporary vector. In order to ensure $\mathbf{q}_{n+1}$ is orthonormal, $h_{n+1,n} = \|\mathbf{v}\|$. In this paper, $\|\cdot\|$ will indicate a 2-norm. The Arnoldi iteration is presented in Algorithm 2. Since the Arnoldi iteration is used for eigenvalue calculations as well, $b$ will be considered an arbitrary vector. Lines 5 and 6 of Algorithm 2 perform the operations in Eq. (6.7). Also in the algorithm on line 2, the loop can go for an arbitrary number of iterations. This iteration parameter is specified by the user and is problem-specific. Therefore, with each Arnoldi iteration, projections are made onto successive Krylov subspaces.

## 6.2 Generalized Minimal RESidual method (GMRES)

The Arnoldi iteration that was presented in the previous section is used to find eigenvalues of a system. GMRES, on the other hand, can be used to solve $\mathbb{A}\mathbf{x} = \mathbf{b}$. According to Trefethen, the idea behind GMRES is that at iteration step $n$, $\mathbf{x}$ is approximated with $\mathbf{x}_n \in \mathcal{K}_n$ that minimizes the norm of residual $\mathbf{r}_n = \mathbf{b} - \mathbb{A}\mathbf{x}_n$. Therefore, $\mathbf{x}_n$ is determined by solving a least squares problem. To solve this, the following Krylov matrix is constructed

$$\mathbb{A}\mathbb{K}_n = \left[ \begin{array}{c|c|c|c} \mathbb{A}\mathbf{b} & \mathbb{A}^2\mathbf{b} & \cdots & \mathbb{A}^n\mathbf{b} \end{array} \right]. \tag{6.9}$$

The least squares problem then becomes

$$\|\mathbb{A}\mathbb{K}_n\mathbf{c} - \mathbf{b}\| = \text{minimum}, \tag{6.10}$$

**Algorithm 2** Arnoldi Iteration [14, 17]

```
1: $b$ =arbitrary, $q_1 = b/\|b\|$
2: for $n = 1, 2, 3, ...$ do
3:     $v = \mathbf{A}q_n$
4:     for $j = 1..n$ do
5:         $h_{jn} = q_j^* v$
6:         $v = v - h_{jn}q_j$
7:     end for
8:     $h_{n+1,n} = \|v\|$
9:     $q_{n+1} = v/h_{n+1,n}$
10: end for
```

where $c$ is determined such that the 2-norm of the residual is minimized. It can be seen that $\mathbf{x}_n = \mathbb{K}_n\mathbf{c}$. Solving the least squares problem is discussed in Section 6.2.2. One method to solve this problem is to use QR factorization of $\mathbb{A}\mathbb{K}_n$. According to Trefethen [17], this approach is numerically unstable and generates a matrix $\mathbb{R}$ which is not utilized. Instead, the Arnoldi iteration from Algorithm 2 is used to generate a sequence of Krylov matrices denoted by $\mathbb{Q}_n$ whose columns span the Krylov subspace $\mathcal{K}_n$. Therefore, Eq. (6.10) can be rewritten as

$$\|\mathbb{A}\mathbb{Q}_n\mathbf{y} - \mathbf{b}\| = \text{minimum}, \tag{6.11}$$

so that $\mathbf{x}_n = \mathbb{Q}_n\mathbf{y}$. Equation (6.5) can be used to rewrite Eq. (6.11),

$$\left\|\mathbb{Q}_{n+1}\widetilde{\mathbb{H}}_n\mathbf{y} - \mathbf{b}\right\| = \text{minimum}. \tag{6.12}$$

Since $\mathbb{Q}_{n+1}$ is unitary and the vectors inside the norm are in the column space of this matrix, Eq. (6.12) can be written equivalently as

$$\left\|\widetilde{\mathbb{H}}_n\mathbf{y} - \mathbb{Q}^*_{n+1}\mathbf{b}\right\| = \text{minimum}, \tag{6.13}$$

where $\mathbb{Q}^*_{n+1}$ is the conjugate transpose of $\mathbb{Q}_{n+1}$. Another property of this expression is that $\mathbb{Q}^*_{n+1}\mathbf{b} = \|\mathbf{b}\|\,\mathbf{e}_1$ where $\mathbf{e}_1 = \langle 1, 0, 0...\rangle^*$ . Finally, the GMRES problem can be cast as

$$\left\|\widetilde{\mathbb{H}}_n\mathbf{y} - \|\mathbf{b}\|\,\mathbf{e}_1\right\| = \text{minimum}. \tag{6.14}$$

After the residual norm is below a certain value, the solution can be found with

$$\mathbf{x} = \mathbb{Q}_n\mathbf{y}, \tag{6.15}$$

where here $\mathbb{Q}_n$ is the Krylov matrix determined from the Arnoldi iteration.

### 6.2.1 GMRES Algorithm

The basic GMRES algorithm using the Arnoldi method is listed in Algorithm 3, where the least squares problem is listed as a high level command. This algorithm has a slightly different form as it is written in Saad's notation rather than Trefethen's described above. The main difference is the notation and method used to solve the least squares problem. Saad defines the approximate solution of $x$ with $x_1 + z$, where $x_1$ is some guess of the solution input to the algorithm. The least squares problem is then cast into the form

$$\min \|\mathbf{b} - \mathbb{A}\left[\mathbf{x}_1 + \mathbf{z}\right]\| = \min \|\mathbf{r}_1 - \mathbb{A}\mathbf{z}\| \tag{6.16}$$

**Algorithm 3** Basic GMRES [14]

```
1: $r_1 = b - \mathbf{A}x_1$
2: $q_1 = r_1/\|r_1\|$
3: for $n = 1, 2, 3, ...$ do
4:     $v = \mathbf{A}q_n$
5:     for $j = 1..n$ do
6:         $h_{jn} = q_j^* v$
7:         $v = v - h_{jn}q_j$
8:     end for
9:     $h_{n+1,n} = \|v\|$
10:    $q_{n+1} = v/h_{n+1,n}$
11:    Find $y$ to minimize $\left\|\widetilde{\mathbf{H}}_n y - \beta e_1\right\|$ {where $\beta = \|r_1\|$}
12:    $x_n = \mathbf{Q_n}y$
13: end for
```

so that $\mathbf{z} = \mathbb{Q}_n\mathbf{y}$. Following the same procedure listed above for Trefethen's notation, the least squares problem can be cast in the form,

$$\min \left\| \beta\mathbf{e}_1 - \widetilde{\mathbb{H}}_n\mathbf{y} \right\|, \tag{6.17}$$

where $\beta = \|\mathbf{r}_1\|$. The basic GMRES algorithm has also been extended to incorporate a restart feature. This feature is straightforward and presented in Algorithm 4.

### 6.2.2 Solving the Least Squares Problem

In linear algebra, a least squares problem must be solved if $\mathbb{A}\mathbf{x} = \mathbf{b}$ is overdetermined [17]. One of the methods used to solve a least squares problem is QR factorization. This QR factorization is performed using Gram-Schmidt or Householder triangularization such that $\mathbb{A} = \mathcal{QR}$. Note that this $\mathcal{QR}$ is written in scripts so as to not conflict with $\mathbb{Q}$ from the Arnoldi iteration. To do this, the orthogonal projector $\mathbb{P} = \mathcal{QQ}^*$is applied to $\mathbf{b}$,

$$\mathbf{y} = \mathbb{P}\mathbf{b} = \mathcal{QQ}^*\mathbf{b}. \tag{6.18}$$

The system

$$\mathbb{A}\mathbf{x} = \mathbf{y} \tag{6.19}$$

has an exact solution. Substituting the QR factorization,

$$\mathcal{QR}\mathbf{x} = \mathcal{QQ}^*\mathbf{b}. \tag{6.20}$$

Since $\mathcal{Q}^*\mathbb{A} = \mathcal{R}$, left multiplication of $\mathcal{Q}^*$ gives

$$\mathcal{R}\mathbf{x} = \mathcal{Q}^*\mathbf{b}. \tag{6.21}$$

The left hand side is now an upper triangular matrix and can be solved via back substitution. In the GMRES problem, $\widetilde{\mathbb{H}}_n$ is the coefficient matrix $\mathbf{A}$ shown above with $\beta\mathbf{e}_1$ as $\mathbf{b}$. A practical implementation of solving the least squares problem is to factor $\widetilde{\mathbb{H}}_n$ into $\mathcal{Q}_n\mathcal{R}_n$ using plane rotations (Givens rotation) [14].

### 6.2.3 Introduction to Givens Rotation

It is desirable to not have to perform QR factorization at every iteration of GMRES. Because of the special structure of $\widetilde{\mathbb{H}}_n$, it can be progressively updated at each iteration. Before discussing how to formulate the GMRES algorithm with Givens rotation to perform this task, this method will be applied to an arbitrary matrix.

In general, QR factorizations can be computed with successive Givens rotations. With each rotation, subdiagonal matrix elements are zeroed out. The final matrix is the upper triangular matrix $\mathcal{R}$ while the matrix $\mathcal{Q}$ is given by multiplying the conjugate transpose of all rotation matrices together. In general, a rotation matrix is given by

$$\mathbb{G} = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & ⋰ & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \vdots & s & \cdots & c & \cdots & 0 \\ \vdots & ⋰ & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}, \tag{6.22}$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$ [11]. For example, consider the following coefficient matrix

**Algorithm 4** GMRES w/ Restart [14]

```
1: for $k = 1..max$ do
2:   $r = b - \mathbf{A}x$
3:   $\beta = \|r\|$
4:   if $\beta < tol$ then
5:     leave loop
6:   end if
7:   $q = r/\beta$
8:   for $n = 1...res$ do
9:     $v = \mathbf{A}q_n$
10:    for $j = 1..n$ do
11:      $h_{jn} = q_j^* v$
12:      $v = v - h_{jn}q_j$
13:    end for
14:    $h_{n+1,n} = \|v\|$
15:    $q_{n+1} = v/h_{n+1,n}$
16:    Find $y$ to minimize $\left\|\widetilde{\mathbf{H}}_n y - \beta e_1\right\|$ {where $\beta = \|r_1\|$}
17:  end for
18:  $x_n = \mathbf{Q_n}y$
19: end for
```

$$\mathbb{A} = \begin{bmatrix} 6 & 5 & 0 \\ 5 & 1 & 4 \\ 0 & 4 & 3 \end{bmatrix}. \tag{6.23}$$

Using MATLAB, **A** has the following QR factorization,

$$\mathcal{Q} = \begin{bmatrix} -0.7682 & 0.03327 & -0.5470 \\ -0.6402 & -0.3992 & 0.6564 \\ 0 & 0.8544 & 0.5196 \end{bmatrix} \tag{6.24}$$

and

$$\mathcal{R} = \begin{bmatrix} -7.8102 & -4.4813 & -2.5607 \\ 0 & 4.6817 & 0.9664 \\ 0 & 0 & 4.1843 \end{bmatrix}. \tag{6.25}$$

Looking at the matrix $\mathbb{A}$ in order to triangularize it, the element (2,1) and (3,2) must be eliminated. Taking element (2,1) to be rotated first, the following rotation matrix is constructed,

$$\mathbb{G} = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{6.26}$$

Therefore, in order to compute the appropriate $c$ and $s$ values, the following system of equations is solved

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}. \tag{6.27}$$

In this problem, it is known that a zero needs to be replaced in the target element. Here it is element $a_{21} = 5$. In addition to the above equation, it is also known that $c^2 + s^2 = 1$ as explained in the definition of the rotation matrix. Combining the following formulas:

$$ca_{11} - sa_{21} = r, \tag{6.28}$$

$$sa_{11} + ca_{21} = 0 \tag{6.29}$$

and

$$c^2 + s^2 = 1, \tag{6.30}$$

gives us an expression for $r$, $c$ and $s$ in terms of $a_{11}$ and $a_{21}$.

$$r^2 = a_{11}^2 + a_{21}^2, \tag{6.31}$$

$$c = \frac{a_{11}}{\sqrt{a_{11}^2 + a_{21}^2}}, \tag{6.32}$$

and

$$s = -\frac{a_{21}}{\sqrt{a_{11}^2 + a_{21}^2}}. \tag{6.33}$$

The parameters $r$, $c$ and $s$ are computed for this example as

$$r = 7.8102,$$

$$c = 0.7682,$$

and

$$s = -0.6402.$$

Apply the rotation matrix $\mathbb{G}$, with $c$ and $s$ now computed, to the coefficient matrix $\mathbf{A}$, it becomes

$$\mathbb{A}' = \mathbb{G}\mathbb{A} = \begin{bmatrix} 7.8102 & 4.4813 & 2.5607 \\ 0 & -2.4327 & 3.0729 \\ 0 & 4 & 3 \end{bmatrix}. \tag{6.34}$$

A few observations can be made. First, the rotation matrix affects only the two rows it is applied to and will subsequently affect all columns in those rows. The third row in this case was untouched. The next rotation matrix will be used to eliminate element (3,2). Therefore, using this element and element (2,2), the rotation matrix will be of the form

$$\mathbb{G}' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix} \tag{6.35}$$

where $r = 4.6817$, $c = -0.5196$ and $s = -0.8544$. Applying the rotation matrix to $\mathbb{A}'$ it becomes

$$\mathbb{A}'' = \mathbb{G}'\mathbb{A}' = \begin{bmatrix} 7.8102 & 4.4813 & 2.5607 \\ 0 & 4.6817 & 0.9664 \\ 0 & 0 & -4.1843 \end{bmatrix}. \tag{6.36}$$

At this point the QR factorization is complete where $\mathcal{R} = \mathbb{A}''$ and

$$\mathcal{Q} = \mathbb{G}^*\mathbb{G}'^* = \begin{bmatrix} 0.7682 & 0.3327 & 0.5470 \\ 0.6402 & -0.3992 & -0.6564 \\ 0 & 0.8544 & -0.5196 \end{bmatrix}. \tag{6.37}$$

As can be observed, the matrices yield the same results (except for negatives) as using the QR factorization routine in MATLAB directly. Thus, successful QR factorization has been shown using a series of Givens rotation matrices. Note that in the least squares problem explained in Section 6.2.2, the conjugate transpose of $\mathcal{Q}$ must be applied to the right hand side vector (see Eq. (6.21)). Therefore, the product of rotation matrices is needed and not the product of their conjugate transposes.

### 6.2.4 Implementation of Givens Rotation in GMRES

Recall the least squares problem from Eq. (6.2.2). A matrix $\mathcal{Q}_n$ with dimensions $(n+1) \times (n+1)$ can be defined such that it is the accumulated product of the conjugate transpose of rotation matrices ($\mathcal{Q}_2 = \mathbb{G}_1^* \mathbb{G}_2^*$). Recall this matrix is unitary,

$$\min \left\| \beta \mathbf{e}_1 - \widetilde{\mathbb{H}}_n \mathbf{y} \right\| = \min \left\| \mathbb{Q}_n \left[ \beta \mathbf{e}_1 - \widetilde{\mathbb{H}}_n \mathbf{y} \right] \right\| = \min \left\| \mathbf{g}_n - \mathcal{R}_n \mathbf{y} \right\| \tag{6.38}$$

where $\mathbf{g}_n \equiv \mathbb{Q}_n \beta \mathbf{e}_1$. The QR factorization for the Hessenberg matrix $\widetilde{\mathbb{H}}_n \in \mathbb{C}^{(n+1)\times n}$ is

$$\widetilde{\mathbb{H}}_n = \mathcal{Q}_n \mathcal{R}_n. \tag{6.39}$$

Left multiplying by the conjugate transpose of this equation becomes,

$$\mathcal{Q}_n^* \widetilde{\mathbb{H}}_n = \mathcal{R}_n. \tag{6.40}$$

As explained after Eq. (6.37), the conjugate transpose of $\mathcal{Q}_n$ in the QR factorization is the accumulated product of rotation matrices,

$$\mathcal{Q}_n^* = \mathbb{G}_n \times \ldots \times \mathbb{G}_2 \times \mathbb{G}_1 \tag{6.41}$$

Thus,

$$\mathbb{Q}_n = \mathcal{Q}_n^*. \tag{6.42}$$

Therefore, Eq. (6.39) can be applied to Eq. (6.38) so that $\mathcal{R}_n = \mathbb{Q}_n \widetilde{\mathbb{H}}_n$. To summarize the above explanation, the least squares problem that is being solved is

$$\min \left\| g_n - \mathcal{R}_n y \right\|, \tag{6.43}$$

where $g_n = \mathbb{Q}_n \beta \mathbf{e}_1$ and $\mathcal{R}_n = \mathbb{Q}_n \widetilde{\mathbb{H}}_n$. It can be seen that the accumulated product of Givens rotation matrices must be applied at each step to $\beta \mathbf{e}_1$ and the Hessenberg matrix $\widetilde{\mathbb{H}}_n$. It would seem then at every step $n$, one would need to compute a new Givens rotation matrix, apply it to the accumulated $\mathbb{Q}_n$ matrix and apply that new matrix to $\beta \mathbf{e}_1$ and $\widetilde{\mathbb{H}}_n$ to compute $\mathbf{g}_n$ and $\mathcal{R}_n$. Due to the structure of $\beta \mathbf{e}_1$, $\widetilde{\mathbb{H}}_n$ and the Givens rotation matrix, this process can be simplified. After the first Arnoldi step in the GMRES algorithm, $\beta \mathbf{e}_1$ and $\widetilde{\mathbb{H}}_n$ are

$$\beta \mathbf{e}_1 = \begin{bmatrix} \beta \\ 0 \end{bmatrix} \qquad \widetilde{\mathbb{H}}_1 = \begin{bmatrix} h_{11} \\ h_{21} \end{bmatrix}. \tag{6.44}$$

After a Givens rotation,

$$\mathbb{G}_1 = \begin{bmatrix} c_1 & -s_1 \\ s_2 & c_2 \end{bmatrix}, \tag{6.45}$$

$\mathbf{g}$ and $\mathcal{R}_n$ are

$$\mathbf{g}^{(1)} = \begin{bmatrix} g_1 \\ g_2 \end{bmatrix} \qquad \mathcal{R}_n = \begin{bmatrix} r_{11} \end{bmatrix}. \tag{6.46}$$

On the next iteration,

$$\beta \mathbf{e}_1 = \begin{bmatrix} \beta \\ 0 \\ 0 \end{bmatrix} \qquad \widetilde{\mathbb{H}}_2 = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \\ & h_{23} \end{bmatrix}. \tag{6.47}$$

After two Givens, the first with

$$\mathbb{G}_1 = \begin{bmatrix} c_1 & -s_1 & 0 \\ s_1 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.48}$$

and the second

$$\mathbb{G}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_1 & -s_1 \\ 0 & s_1 & c_1 \end{bmatrix}, \tag{6.49}$$

$g$ and $\mathbf{R}_n$ are

$$\mathbf{g}^{(2)} = \begin{bmatrix} g_1 \\ g_2' \\ g_3 \end{bmatrix} \qquad \mathcal{R}_n = \begin{bmatrix} r_{11} & r_{12} \\ 0 & r_{22} \end{bmatrix}. \tag{6.50}$$

What is interesting to observe is that by doing this, the Givens rotation applied in the previous iteration is performed again such that $g_1$ and $r_{11}$ are the same as before. This is because the Hessenberg matrix is all zeros below the first subdiagonal such that future Givens rotations will not affect that column. The same answer could have been calculated with

$$\mathbf{g}^{(2)} = \mathbb{G}_2\mathbf{g}^{(1)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_1 & -s_1 \\ 0 & s_1 & c_1 \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ 0 \end{bmatrix} \tag{6.51}$$

and

$$\mathcal{R}_2^{\langle 2 \rangle} = \mathbb{G}_2\mathbb{G}_1\widetilde{\mathbb{H}}_2^{\langle 2 \rangle}. \tag{6.52}$$

where $\langle \cdot \rangle$ denotes a column. Thus, a recursive relationship can be found for an arbitrary step $n$,

$$\mathbf{g}^{(n)} = \mathbb{G}_n\mathbf{g}^{(n-1)} \tag{6.53}$$

and

$$\mathcal{R}_n^{\langle n \rangle} = \prod_{i=1}^{n} \mathbb{G}_n\widetilde{\mathbb{H}}_n^{\langle n \rangle}. \tag{6.54}$$

This recursive procedure simplifies computations from $O\left(n^2\right)$ to $O\left(n\right)$ since the rotations do not have to be applied to the whole matrix again. Since only two operations need to performed when a Givens rotation is applied, simple computations are performed on individual elements instead of the entire matrix. To save on memory, it can be seen from Eq. (6.54) that the matrix $\mathcal{R}_n$ only depends on column $n$ of the Hessenberg matrix. Looking at Algorithm 3, column $n$ of the Hessenberg matrix does not depend directly on any elements from the previous column. Therefore, the new column of the matrix $\mathcal{R}_n$ computed from Eq. (6.54) can be stored back into the $n$-th column of $\widetilde{\mathbb{H}}_n$. Therefore, the expression can be rewritten as

$$\widetilde{\mathbb{H}}_n^{\langle n \rangle} = \prod_{i=1}^{n} \mathbb{G}_n\widetilde{\mathbb{H}}_n^{\langle n \rangle}. \tag{6.55}$$

This algorithm is presented in Algorithm 5. For the $n$-th column of the Hessenberg matrix computed from the Arnoldi iteration, all the previous Givens rotation parameters are applied. Then, new rotation parameters are calculated and applied to the last two rows of the $n$-th column of the Hessenberg matrix and the vector $g$.

The last question that has not yet been answered is when to stop the GMRES iteration. After iteration $n$, the residual norm is given by

$$\|\mathbf{r}_n\| = \|\mathbf{g}_n - \mathcal{R}_n\mathbf{y}\|. \tag{6.56}$$

Due to the structure of $\mathcal{R}_n$ and $\mathbf{y}$, this norm is equivalent to the absolute value of the $n$-th row in $g$ . The convergence criteria is shown as

$$|g_n| < tol.$$

This linear tolerance is user defined and should be compared to the nonlinear tolerance in the Newton iteration. After this convergence criteria is met, the vector $y$ can be computed via back substitution with

**Algorithm 5** Givens Rotation [14]

```
Require: For step $n$:: $c(1:n-1)$, $s(1:n-1)$, $\widetilde{\mathbf{H}}_n$, and $g(1:n-1)$
 1: for $i = 1..n-1$ do
 2:     $a = c_i h_{in} - s_i h_{i+1,n}$ {apply previous Givens rotations}
 3:     $b = s_i h_{in} + c_i h_{i+1,n}$
 4:     $h_{in} \leftarrow a$
 5:     $h_{i+1,n} \leftarrow b$
 6: end for
 7: $c_n = \frac{h_{nn}}{\sqrt{h_{nn}^2 + h_{n+1,n}^2}}$ {calculate new rotation parameters}
 8: $s_n = \frac{-h_{n+1,n}}{\sqrt{h_{nn}^2 + h_{n+1,n}^2}}$
 9: $a = c_n h_{nn} - s_n h_{n+1,n}$ {apply current Givens rotations}
10: $b = s_n h_{nn} + c_n h_{n+1,n}$
11: $h_{nn} \leftarrow a$
12: $h_{n+1,n} \leftarrow b$
13: $a = c_n g_n - s_n g_{n+1}$
14: $b = s_n g_n + c_n g_{n+1}$
15: $g_n \leftarrow a$
16: $g_{n+1} \leftarrow b$
17: return $c(1:n)$, $s(1:n)$, $\widetilde{\mathbf{H}}_n$, and $g(1:n)$
```

$$\mathbf{g}_n = \mathcal{R}_n \mathbf{y}. \tag{6.57}$$

Finally, the solution vector can be computed with

$$\mathbf{x} = \mathbf{x}_0 + \mathbb{Q}_n \mathbf{y}, \tag{6.58}$$

where $\mathbb{Q}_n$ is the Krylov matrix from the Arnoldi iteration. The GMRES algorithm from Algorithm 3 can now be extended for Givens rotations, presented in Algorithm 6. Source code for this solver is listed in Appendix A.2.

# 7 Inexact Newton's Method and Jacobian-Free Approximation

The previous two sections discussed how to solve nonlinear equations with Newton's method, and how to solve a linear system of equations with GMRES. The combination of these two methods is called Newton-Krylov, or more specifically Newton-GMRES. Since each linear system solved in Newton's iteration is not solved exactly with a direct method, the result from the iterative solver is not exact. This idea is known as Inexact Newton's method and can be taken advantage of.

## 7.1 Inexact Newton's Method

Since the user can set the nonlinear tolerance in the Newton iteration and the linear tolerance in the GMRES solver, these tolerances can be somewhat optimized to each other. Since at the beginning of Newton's iteration, the nonlinear residual is quite large, the linear system does not need to be converged very tightly to get a good approximation of the next nonlinear step [12]. This idea is formalized by making the convergence of the linear residual proportional to the nonlinear residual,

$$\left\|\mathbb{J}\left(\mathbf{x}^n\right) d\mathbf{x}_m^n + \mathbf{F}\left(\mathbf{x}^n\right)\right\|_2 < \eta \left\|\mathbf{F}\left(\mathbf{x}^n\right)\right\|_2. \tag{7.1}$$

Here, $n$ is the nonlinear Newton iteration number, $m$ is the linear iteration number of the Krylov solver and $\eta$ is the relative residual tolerance. To use this appropriately, the user should specify the relative residual tolerance parameter. In the Krylov solver, the nonlinear residual is just the right hand side of $\mathbb{A}\mathbf{x} = \mathbf{b}$ and

**Algorithm 6** GMRES w/Givens Rotations [14]

```
for $k = 1..max$ do
    $r = b - \mathbf{A}x$
    $\beta = \|r\|$
    if $\beta < tol$ then
        leave loop
    end if
    $g = \beta e_1$
    $q = r/\beta$
    for $n = 1...res$ do
        $v = \mathbf{A}q_n$
        for $j = 1..n$ do
            $h_{jn} = q_j^* v$
            $v = v - h_{jn} q_j$
        end for
        $h_{n+1,n} = \|v\|$
        $q_{n+1} = v/h_{n+1,n}$
        perform Givens rotation (see Alg. 5)
        if $|g_n| < tol$ then
            leave loop
        end if
    end for
    solve for y, $g_n = \widetilde{\mathbf{H}}_n y$
    $x_n = \mathbf{Q_n} y$
end for
```

must be given to the Krylov solver. Thus, the absolute linear tolerance in the Krylov solver is just $\eta \|\mathbf{b}\|_2$. When the norm of $\mathbf{b}$ is large, at initial Newton steps, the linear tolerance is not that tight. However, at later Newton steps when the norm of $\mathbf{b}$ is small, the linear tolerance is much tighter to help converge the system. Therefore iterations in the Krylov solver are not wasted when the nonlinear residual is large. This $\eta$ parameter is problem dependent and can be optimized for each problem. Source code for this solver is listed in Appendix A.1.

## 7.2 Jacobian-Free Newton-Krylov Method

At each Newton step, the linear system $\mathbb{J}d\mathbf{x} = -\mathbf{F}$ is solved. From equation (6.1) and (6.58), at the $m$-th Krylov step, the solution of the linear system is

$$d\mathbf{x}_m = d\mathbf{x}_0 + a_0\mathbf{r}_0 + a_1\mathbb{J}\mathbf{r}_0 + a_2\mathbb{J}^2\mathbf{r}_0 + ... + a_m\mathbb{J}^m\mathbf{r}_0, \tag{7.2}$$

where $\mathbf{r}_0$ is the initial linear residual. What is seen from the building of the Krylov subspace is that the Jacobian is always acting on a vector. Even if the Jacobian can be formulated analytically, why use the memory and form the matrix instead of writing a separate routine to perform this multiplication manually? If the Jacobian cannot be formulated analytically, the Jacobian-vector product can be approximated with a finite difference,

$$\mathbb{J}\mathbf{y} \approx \frac{\mathbf{F}(\mathbf{x} + \epsilon\mathbf{y}) - \mathbf{F}(\mathbf{x})}{\epsilon}, \tag{7.3}$$

where $\mathbf{y}$ is an arbitrary vector, $\mathbf{x}$ is the current estimate of the nonlinear solution from Newton's iteration and $\epsilon$ is the perturbation parameter. The choice of the perturbation parameter is arbitrary, but will have an effect on the number of iterations of the problem. In the JFNK overview paper there are suggestions for choosing this perturbation parameter [9]. Mousseau [12] recommends using

$$\epsilon = \frac{\sum_{i=1}^{N} bx_i}{N\|\mathbf{y}\|_2}. \tag{7.4}$$

where $b = 1 \times 10^{-8}$. This is the definition of the perturbation parameter used in this work to approximate the Jacobian-vector product. Even if the Jacobian-vector product can be evaluated analytically, it can also be approximated with a finite difference. In the system of equations solved in this work, all of the Jacobian-vector products can be evaluated analytically except for the density evaluation from the state equation. It must be calculated with a finite difference approximation. The idea of forming the Jacobian-vector products analytically or approximating them with a finite difference is investigated in this work. If the finite difference approximation of the whole system can be performed quicker, then this extra routine to perform the analytic Jacobian-vector product is not needed. This would be ideal because these routines can become complicated to write. Source code for this finite difference approximation is listed in Appendix A.4.

## 7.3 Preconditioning

It is important to keep the number of iterations in the GMRES solver to a minimum. The first reason is that with fewer iterations, each linear step solve will go very quickly. Another important reason is that in GMRES, all previous iterations' Krylov vectors are stored in memory. The term preconditioner means to multiply the coefficient matrix $\mathbb{A}$ by a preconditioner matrix $\mathbb{M}^{-1}$ [7]. This preconditioner can be multiplied on the left, right or on both sides of $\mathbb{A}$. The idea is that the action of the preconditioner on the coefficient matrix will result in an easier linear system solve. In this work only left preconditioning is used where

$$\mathbb{M}^{-1}\mathbb{A}\mathbf{x} = \mathbb{M}^{-1}\mathbf{b}. \tag{7.5}$$

There are many methods to compute a preconditioner for a linear system. Such preconditioners are Jacobi, Incomplete LU factorization (ILU), block preconditioners, multigrid, physics-based, etc. [9, 15].

In this work, ILU preconditioning is used as it is relatively easier to form. An ILU process computes sparse lower and upper triangular matrices ($\mathbb{L}$ and $\mathbb{U}$) such that a residual matrix defined as

$$\mathbb{R} = \mathbb{L}\mathbb{U} - \mathbb{A}, \tag{7.6}$$

is constrained to certain conditions. The simplest constraint is that the product of $\mathbb{L}\mathbb{U}$ has the same number and locations of nonzeros, known as Zero Fill-in ILU. This is used in MATLAB when the function `ilu` with '`nofill`' is used. We therefore have that

$$\mathbb{M} = \mathbb{L}\mathbb{U} \tag{7.7}$$

and

$$\mathbb{U}^{-1}\mathbb{L}^{-1}\mathbb{A}\mathbf{x} = \mathbb{U}^{-1}\mathbb{L}^{-1}\mathbf{b}. \tag{7.8}$$

This can be added to the basic GMRES algorithm, now modified in Algorithm 7.

**Algorithm 7** Basic GMRES w/Preconditioning [14]

```
1: $\mathbf{r}_1 = \mathbb{U}^{-1}\mathbb{L}^{-1}(\mathbf{b} - \mathbb{A}\mathbf{x}_1)$
2: $\mathbf{q}_1 = \mathbf{r}_1/\|\mathbf{r}_1\|$
3: for $n = 1, 2, 3, ...$ do
4:     $\mathbf{v} = \mathbb{U}^{-1}\mathbb{L}^{-1}\mathbb{A}\mathbf{q}_n$
5:     for $j = 1..n$ do
6:         $h_{jn} = q_j^* v$
7:         $v = v - h_{jn} q_j$
8:     end for
9:     $h_{n+1,n} = \|\mathbf{v}\|$
10:    $\mathbf{q}_{n+1} = \mathbf{v}/h_{n+1,n}$
11:    Find $\mathbf{y}$ to minimize $\left\|\widetilde{\mathbb{H}}_n\mathbf{y} - \beta\mathbf{e}_1\right\|$ {where $\beta = \|\mathbf{r}_1\|$}
12:    $\mathbf{x}_n = \mathbb{Q}_n\mathbf{y}$
13: end for
```

# 8 Solving the Steady State Neutron Diffusion Equation

As an introduction to solving this coupled set of nonlinear neutronics and thermal hydraulics equations, a simpler problem of neutronics was solved first. The steady steady neutron diffusion equation to be solved is

$$\mathbb{M}\bar{\mathbf{\Phi}} = \lambda \mathbb{F}\bar{\mathbf{\Phi}}, \tag{8.1}$$

where $\lambda = 1/k_{eff}$. This equation was derived in Section 4.1.1. This linear system of equations is an eigenvalue problem. One straight-forward approach for solving linear eigenvalue problems is to use power iteration. The power iteration method is outlined in Algorithm 8 [3]. The eigenvalue is updated in each iteration with a scalar product of the new fission source with itself divided by the new fission source with the old fission source. This method is widely used and will be used as the "right" answer when we compare it against Newton's method. Another important feature of power iteration is that it will always converge on the fundamental solution of the equation whereas any mode can be obtained when using Newton's method.

In the power iteration, the flux vector is the only unknown and hence the system of equations is linear. However, if the eigenvalue is also considered as an unknown, the system of equations becomes nonlinear. Therefore Newton's method is used to solve for these unknowns. The residual equations are

$$\mathbf{F} = \begin{bmatrix} \mathbb{M}\bar{\mathbf{\Phi}} - \lambda\mathbb{F}\bar{\mathbf{\Phi}} \\ -\frac{1}{2}\bar{\mathbf{\Phi}}^\top\bar{\mathbf{\Phi}} + \frac{1}{2} \end{bmatrix}. \tag{8.2}$$

The last equation represents the unity L2-norm constraint on the eigenvector. The unknown vector is setup as

$$\mathbf{x} = \begin{bmatrix} \bar{\mathbf{\Phi}} \\ \lambda \end{bmatrix}. \tag{8.3}$$

Using these equations, the Jacobian can be constructed as

$$\mathbb{J} = \begin{bmatrix} \mathbb{M} - \lambda\mathbb{F} & -\mathbb{F}\bar{\mathbf{\Phi}} \\ -\bar{\mathbf{\Phi}}^\top & 0 \end{bmatrix}. \tag{8.4}$$

These set of equations can be solved 3 ways: (1) power iteration, (2) construct analytic Jacobian-vector product, (3) approximate Jacobian vector product with finite difference. In method (2), the analytic Jacobian-vector product can be formulated as

$$\mathbb{J}\mathbf{y} = \begin{bmatrix} \mathbb{M} - \lambda\mathbb{F} & -\mathbb{F}\bar{\mathbf{\Phi}} \\ -\bar{\mathbf{\Phi}}^\top & 0 \end{bmatrix} \begin{bmatrix} y_\phi \\ y_\lambda \end{bmatrix} = \begin{bmatrix} (\mathbb{M} - \lambda\mathbb{F})\, y_\phi - \mathbb{F}\bar{\mathbf{\Phi}} y_\lambda \\ -\bar{\mathbf{\Phi}}^\top y_\phi \end{bmatrix}. \tag{8.5}$$

The finite difference approach to the Jacobian-vector product is described in Section 7.2.

Each of these methods were implemented in a MATLAB code which is presented in Appendix B.2. The first case is for a reactor of 600 cm that has a high dominance ratio of 0.997. Note the dominance ratio is the ratio of the two largest eigenvalues of the system [18]. The flux eigenvectors as shown in Fig. 8.1 were compared for each of the three cases. The results show that Newton-based methods do not agree with the power iteration even though the residual criteria is met. This is because any eigenvalue/eigenvector pair

**Algorithm 8** Power Iteration Method

```
1: $\mathbf{\Phi}^{(0)}$ = arbitrary nonzero vector
2: $k_{eff}^{(0)}$ = arbitrary nonzero constant
3: for $n = 1, 2, 3, ...$ do
4:     $\mathbf{b} = 1/k_{eff}^{(n-1)}\mathbb{F}\mathbf{\Phi}^{(n-1)}$
5:     $\mathbf{\Phi}^{(n+1)} = \mathbb{M}^{-1}\mathbf{b}$
6:     $k_{eff}^{(n)} = k_{eff}^{(n-1)}\frac{\left(\mathbb{F}\mathbf{\Phi}^{(n+1)}, \mathbb{F}\mathbf{\Phi}^{(n+1)}\right)}{\left(\mathbb{F}\mathbf{\Phi}^{(n)}, \mathbb{F}\mathbf{\Phi}^{(n+1)}\right)}$
7:     check convergence of eigenvalue and eigenvector
8: end for
```

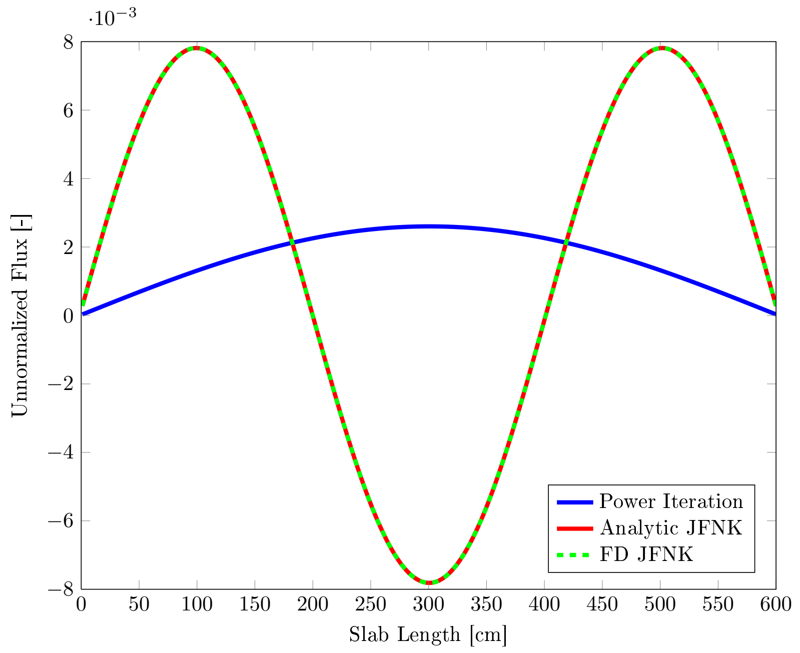


Figure 8.1. Flux Eigenvector Comparison, 600 cm

satisfies the residual equations. To verify, the `eigs` command in MATLAB was used to get an array of eigenvalues of the system. Sure enough, the converged eigenvalue was listed. To ensure that the Newton-based methods converge to the fundamental mode, two power iterations were computed to get the flux shape in the correct direction. After applying this, the results are consistent and are shown in Figs. 8.2 and 8.3.

The differences in the plot, with respect to power iteration, are within the converged nonlinear iteration tolerance of $10^{-6}$. It is more interesting to compare is the number of iterations, the final residual and the computational time. These results are listed in Table 1. From the results, the power iteration takes almost 1000 iterations. This is because the rate of convergence of power iteration is inversely proportional to the dominance ratio. The closer the dominance ratio is to unity, the more iterations it will take to converge. The analytic Jacobian-free method performs the best with respect to computational time. The finite difference version of JFNK does not do as well as the analytic multiplication. However, the time it takes to do this is very comparable to power iteration. Both method (2) and method (3) will be compared again in the coupled neutrons/thermal hydraulics problem.

The slab width was then decreased to 370 cm which is about the length of the active fuel rod length producing power in a standard pressurized water reactor. The dominance ratio of this system is 0.992. The difference in the flux eigenvectors is shown in Fig. 8.4. A comparison of the number of iterations, final residual and computation time is presented in Table 2. From the results, the time to perform the power iteration method takes fewer iterations than in the high dominance ratio case. Therefore, it is observed that the dominance ratio of the system has a direct effect on how long the power iteration takes. Both JFNK methods took the same amount of iterations, with the finite difference approach taking slightly longer to

Table 1. Comparison of Methods to Solve Neutronic Eigenvalue Problem, 600 cm

| Method | Iterations | Final Residual | Time [s] |
|---|---|---|---|
| Power Iteration | 984 | $9.9997 \times 10^{-7}$ | 0.116 |
| Analytic JFNK | 6 | $5.024 \times 10^{-7}$ | 0.075 |
| Finite Difference JFNK | 6 | $4.2829 \times 10^{-7}$ | 0.147 |

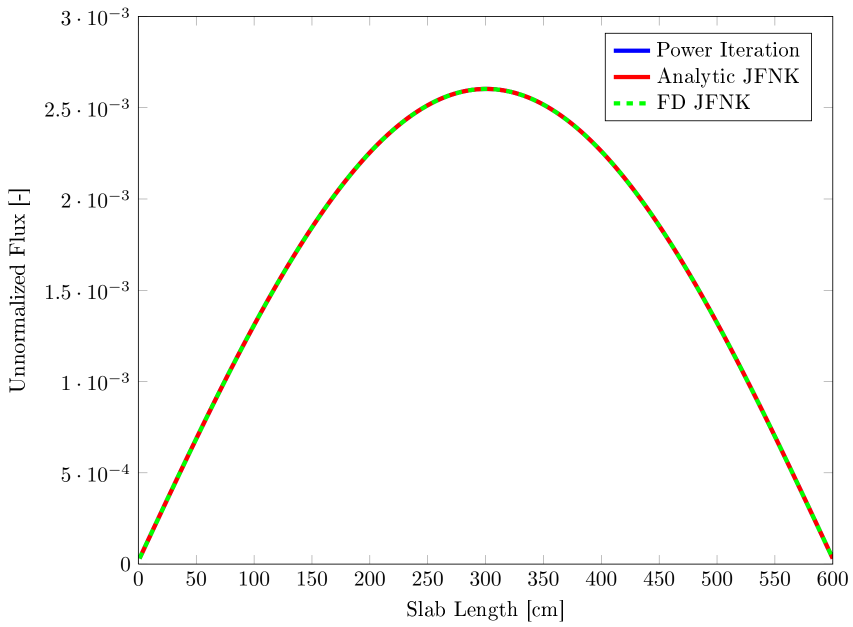


Figure 8.2. Comparison of Flux with Initial Power Iteration, 600 cm

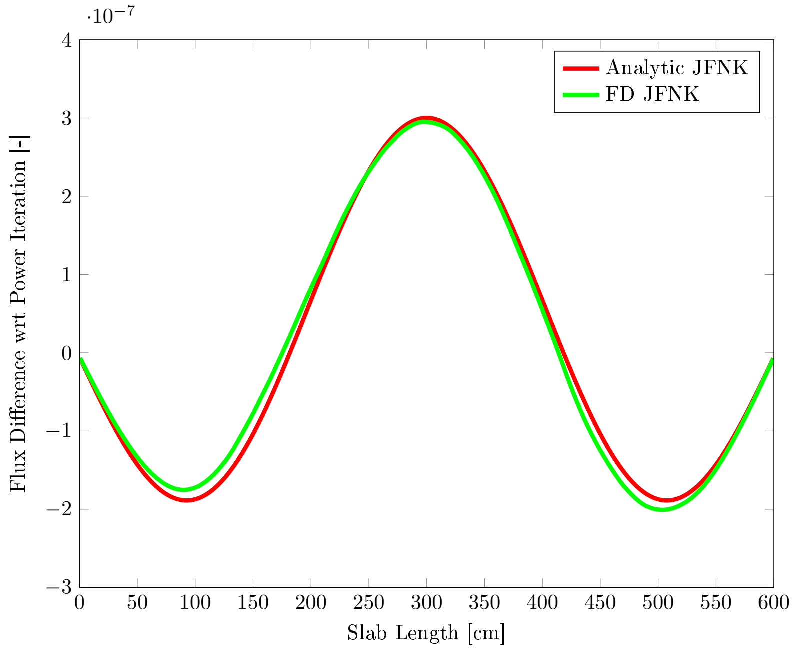


Figure 8.3. Difference of Flux with Initial Power Iteration, 600 cm

Table 2. Comparison of Methods to Solve Neutronic Eigenvalue Problem, 300 cm

| Method | Iterations | Final Residual | Time [s] |
|---|---|---|---|
| Power Iteration | 424 | $9.9881 \times 10^{-7}$ | 0.07037 |
| Analytic JFNK | 5 | $5.024 \times 10^{-7}$ | 0.05714 |
| Finite Difference JFNK | 5 | $4.2829 \times 10^{-7}$ | 0.08562 |

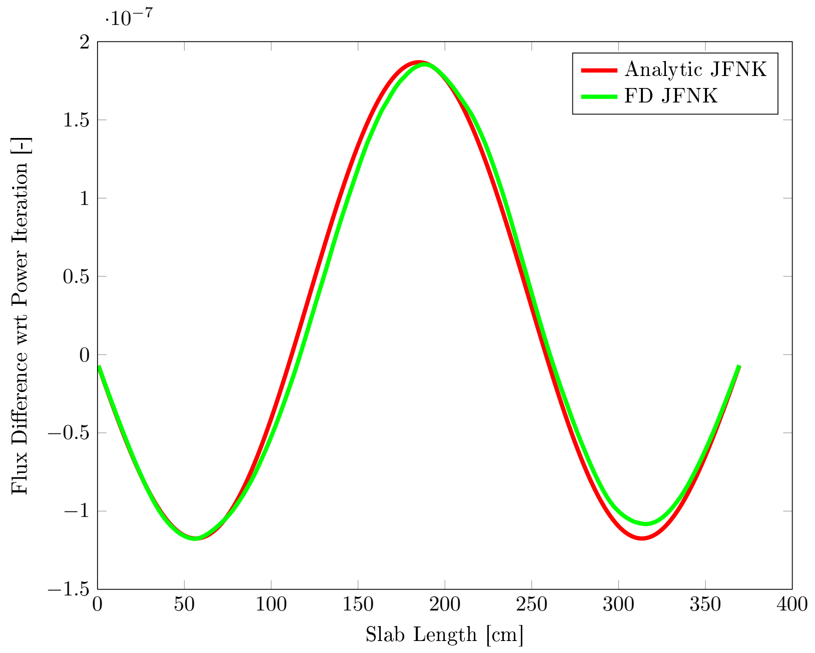


Figure 8.4. Flux Eigenvector Comparison, 300 cm

solve. This is due to the number of GMRES inner iterations needed to converge the system. The JFNK method was therefore successfully applied to the neutronics eigenvalue problem.

## 9 Steady State Coupled Physics Solution

In this section the determination of the steady state spatial distribution of flux, power temperature and density will be determined. Equations (4.25), (4.26), (4.29), (4.36), (4.38), (4.43), (4.44), (4.45) and (4.46) comprise the set of equations needed to solve for the steady distributions. The residual equations can then be formulated as

$$\mathbf{F} = \begin{bmatrix} \mathbb{M}\mathbf{\Phi} - \lambda\mathbb{F}\mathbf{\Phi} \\ Q_R - \tilde{c}\kappa\mathbf{\Sigma}_f^{\mathrm{T}}\mathbf{\Phi}\Delta x. \\ \mathbf{Q} - \tilde{c}\mathbb{E}\mathbf{\Phi}\Delta x \\ \mathbb{S}\mathbf{T} - \mathbb{R}\mathbf{Q} \\ \mathcal{P} - \rho\left(\mathbf{T}, p\right) \\ \mathbf{\Sigma}_a - \Sigma_a^{ref} - \frac{\partial \Sigma_a}{\partial \rho}\left[\mathcal{P} - \rho^{ref}\right] \\ \nu\mathbf{\Sigma}_f - \nu\Sigma_f^{ref} - \frac{\partial \nu\Sigma_f}{\partial \rho}\left[\mathcal{P} - \rho^{ref}\right] \\ \mathbf{D} - D^{ref} - \frac{\partial D}{\partial \rho}\left[\mathcal{P} - \rho^{ref}\right], \\ \kappa\mathbf{\Sigma}_f - \kappa\Sigma_f^{ref} - \frac{\partial \kappa\Sigma_f}{\partial \rho}\left[\mathcal{P} - \rho^{ref}\right] \\ -\frac{1}{2}\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi} + \frac{1}{2} \end{bmatrix}. \tag{9.1}$$

Note that in the nonlinear equations, the operators $\mathbb{M}$, $\mathbb{F}$ and $\mathbb{E}$ have to updated since they depend on neutronic parameters. These residual equations are written in a function in MATLAB and can be called during the Newton iteration. The unknown vector is then constructed as

$$\mathbf{x} = \begin{bmatrix} \mathbf{\Phi} \\ \tilde{c} \\ \mathbf{Q} \\ \mathbf{T} \\ \mathcal{P} \\ \mathbf{\Sigma}_a \\ \nu\mathbf{\Sigma}_f \\ \mathbf{D} \\ \kappa\mathbf{\Sigma}_f \\ \lambda \end{bmatrix}. \tag{9.2}$$

From the residual vector and unknown vector, the analytic Jacobian-vector product can be constructed in matrix notation as

$$\mathbb{J}\mathbf{y} = \left[\begin{array}{c|c|c|c|c|c|c|c|c|c}
\mathbb{M}-\lambda\mathbb{F} & 0 & 0 & 0 & 0 & \text{diag}\left\{\mathbf{\Phi}\right\} & -\lambda\text{diag}\left\{\mathbf{\Phi}\right\} & \mathbb{MD} & 0 & -\mathbb{F}\mathbf{\Phi} \\ \hline
-\tilde{c}\kappa\mathbf{\Sigma}_f^{\mathrm{T}}\Delta x & -\kappa\mathbf{\Sigma}_f^{\mathrm{T}}\mathbf{\Phi}\Delta x & 0 & 0 & 0 & 0 & 0 & 0 & -\tilde{c}\mathbf{\Phi}^{\mathrm{T}}\Delta x & 0 \\ \hline
-\mathbb{E}\tilde{c}\Delta x & -\mathbb{E}\mathbf{\Phi}\Delta x & \mathbb{I} & 0 & 0 & 0 & 0 & 0 & -\tilde{c}\Delta x\text{diag}\left\{\mathbf{\Phi}\right\} & 0 \\ \hline
0 & 0 & -\mathbb{R} & \mathbb{S} & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & -\rho\left(T\right)* & \mathbb{I} & 0 & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & -\frac{\partial\Sigma_a}{\partial\rho}\mathbb{I} & \mathbb{I} & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & -\frac{\partial\nu\Sigma_f}{\partial\rho}\mathbb{I} & 0 & \mathbb{I} & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & -\frac{\partial D}{\partial\rho}\mathbb{I} & 0 & 0 & \mathbb{I} & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & -\frac{\partial\Sigma_a}{\partial\rho}\mathbb{I} & 0 & 0 & 0 & \mathbb{I} & 0 \\ \hline
-\mathbf{\Phi}^{\mathrm{T}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right] \begin{bmatrix} y_\phi \\ y_{\tilde{c}} \\ y_Q \\ y_T \\ y_\rho \\ y_{\Sigma_a} \\ y_{\nu\Sigma_f} \\ y_D \\ y_{\kappa\Sigma_f} \\ y_\lambda \end{bmatrix}. \tag{9.3}$$

Performing the Jacobian-vector product analytically, the set of equations is

$$\mathbb{J}\mathbf{y} = \begin{bmatrix}
\left(\mathbb{M}-\lambda\mathbb{F}\right)y_\phi + \left(\text{diag}\left\{\mathbf{\Phi}\right\}\right)y_{\Sigma_a} + \left(-\lambda\text{diag}\left\{\mathbf{\Phi}\right\}\right)y_{\nu\Sigma_f} + \left(\mathbb{MD}\right)y_D + \left(-\mathbb{F}\mathbf{\Phi}\right)y_\lambda \\
\left(-\tilde{c}\kappa\mathbf{\Sigma}_f^{\mathrm{T}}\Delta x\right)y_\phi + \left(-\kappa\mathbf{\Sigma}_f^{\mathrm{T}}\mathbf{\Phi}\Delta x\right)y_{\tilde{c}} + \left(-\tilde{c}\mathbf{\Phi}^{\mathrm{T}}\Delta x\right)y_{\kappa\Sigma_f} \\
\left(-\mathbb{E}\tilde{c}\Delta x\right)y_\phi + \left(-\mathbb{E}\mathbf{\Phi}\Delta x\right)y_{\tilde{c}} + \left(\mathbb{I}\right)y_Q + \left(-\tilde{c}\Delta x\text{diag}\left\{\mathbf{\Phi}\right\}\right)y_{\kappa\Sigma_f} \\
\left(-\mathbb{R}\right)y_Q + \left(\mathbb{S}\right)y_T \\
\left[-\rho\left(T\right)*\right]y_T + \left(\mathbb{I}\right)y_\rho \\
\left(-\frac{\partial\Sigma_a}{\partial\rho}\mathbb{I}\right)y_\rho + \left(\mathbb{I}\right)y_{\Sigma_a} \\
\left(-\frac{\partial\nu\Sigma_f}{\partial\rho}\mathbb{I}\right)y_\rho + \left(\mathbb{I}\right)y_{\nu\Sigma_f} \\
\left(-\frac{\partial D}{\partial\rho}\mathbb{I}\right)y_\rho + \left(\mathbb{I}\right)y_D \\
\left(-\frac{\partial\kappa\Sigma_f}{\partial\rho}\mathbb{I}\right)y_\rho + \left(\mathbb{I}\right)y_{\kappa\Sigma_f} \\
\left(-\mathbf{\Phi}^{\mathrm{T}}\right)y_\phi
\end{bmatrix} \tag{9.4}$$

This Jacobian-vector operation is performed in a MATLAB function that can be called upon by the GMRES solver. In Eqs. (9.3) and (9.4):

- diag{} is the diagonal operator that places a vector along the diagonal of a matrix,
- $\mathbb{MD}$ is the partial derivative of the operator $\mathbb{M}$ with respect to the diffusion coefficient,
- $\mathbb{I}$ is the identity matrix,
- $\left[-\rho\left(T\right)*\right]y_T$ is the partial derivative of the state equation with respect to temperature and must be handled with a finite difference.

The $\mathbb{MD}$ operator has the following form for the boundaries and interior cells:

– Left boundary

$$\left(\frac{2}{\Delta x^2}\frac{D_2^2}{\left(D_2+D_1\right)^2}+2\frac{\left(1-\beta\right)^2}{\left[4D_1\left(1+\beta\right)+\Delta x\left(1-\beta\right)\right]^2}\right)\bar{\phi}_1-\frac{2}{\Delta x^2}\frac{D_2^2}{\left(D_2+D_1\right)^2}\bar{\phi}_2, \tag{9.5}$$

– Interior

$$-\frac{2}{\Delta x^2}\frac{D_{i-1}^2}{\left(D_i+D_{i-1}\right)^2}\bar{\phi}_{i-1}+\left(\frac{2}{\Delta x^2}\frac{D_{i+1}^2}{\left(D_{i+1}+D_i\right)^2}+\frac{2}{\Delta x^2}\frac{D_{i-1}^2}{\left(D_i+D_{i-1}\right)^2}\right)\bar{\phi}_i-\frac{2}{\Delta x^2}\frac{D_{i+1}^2}{\left(D_{i+1}+D_i\right)^2}\bar{\phi}_{i+1}, \tag{9.6}$$

– Right boundary

$$-\frac{2}{\Delta x^2}\frac{D_{I-1}^2}{\left(D_I+D_{I-1}\right)^2}\bar{\phi}_{I-1}+\left(2\frac{\left(1-\beta\right)^2}{\left[4D_I\left(1+\beta\right)+\Delta x\left(1-\beta\right)\right]^2}+\frac{2}{\Delta x^2}\frac{D_{I-1}^2}{\left(D_I+D_{I-1}\right)^2}\right)\bar{\phi}_I. \tag{9.7}$$

Each of these equations will be placed on the diagonal of the $\mathbb{MD}$ matrix. The last tricky part is that the partial derivative of the state equation is not known. Therefore, this section will have to be evaluated with a finite difference and has the form,

$$\left[-\rho\left(T\right)*\right]y_T=\frac{\left[\mathcal{P}-\rho\left(\mathbf{T}+\epsilon y_T,p\right)\right]-\left[\mathcal{P}-\rho\left(\mathbf{T},p\right)\right]}{\epsilon}. \tag{9.8}$$

The overall algorithm to solve this problem is presented in Algorithm 9. Once the user-defined input file is set, the code runs a few power iterations to get the gross shape of the flux correct without feedback for an initial guess to the Newton solver. An initial guess of the thermal hydraulics is also computed based on this guessed flux shape. In order to minimize the number of inner iterations in the GMRES solver, a preconditioner must be determined. Since the Jacobian is never formed in the Newton iteration loop, this preconditioner cannot be computed on-the-fly. Also, it would be expensive to recompute a preconditioner at every Newton step. Therefore, the preconditioner is only formed once using the initial guess values. Here, the approximate Jacobian is fully constructed and ILU factorization is performed. This constant preconditioner is then used in the GMRES solver at every Newton iteration.

The solution of this steady state problem took about 7.06 seconds to complete, with 8 total Newton iterations and about 21 inner GMRES iterations on average. The results of this calculation are presented in Figs. 9.1 and 9.2.

The results prove that the feedback is working correctly. Instead of getting a cosine shape as with the case of no feedback (see Section 8), the cosine shape is now skewed to the left. This is due to the density feedback, where the density of water is higher in the left part of the slab than in the right. Therefore, more fissions will occur to the left and more power is produced.

The solution of this problem was not easy. If the above equations are implemented as is, the MATLAB code will not converge on a solution. This is primarily due to the fact that the kappa-fission cross section is so small. It is on the order of $10^{-13}$ which is much smaller than other quantities in the problem. Therefore, the normalization constant is very large to scale the flux eigenvector. This makes the problem very ill-conditioned even though a preconditioner is used. Since the normalization constant is always multiplied by the kappa-fission cross section in the residual equations, another constant of $10^{12}$ is applied in the input file. Therefore, the span of the magnitudes of the flux-to-power normalization constant and kappa-fission cross

**Algorithm 9** Steady State Coupled Solution

```
1: read input
2: perform small number of power iterations to estimate flux shape
3: compute thermal hydraluics based off flux shape
4: form approximate Jacobian and perform ILU to get preconditioner
5: perform Newton Iterations
```

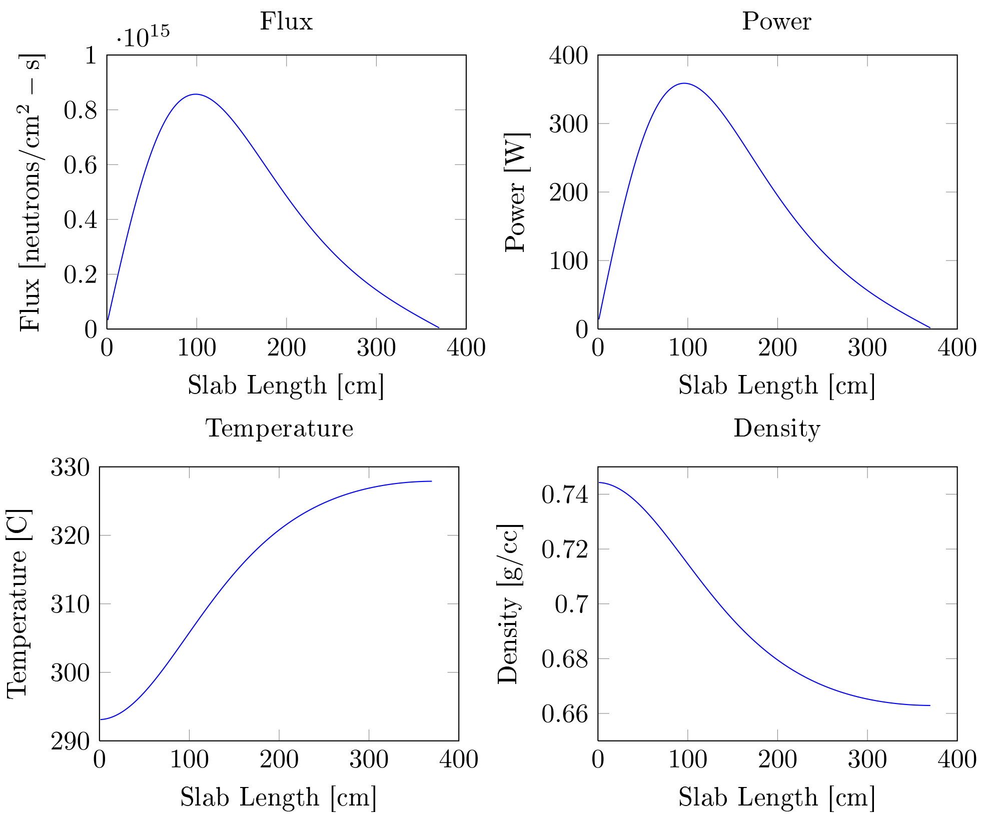


Figure 9.1. Results from Steady State Calculation Part 1

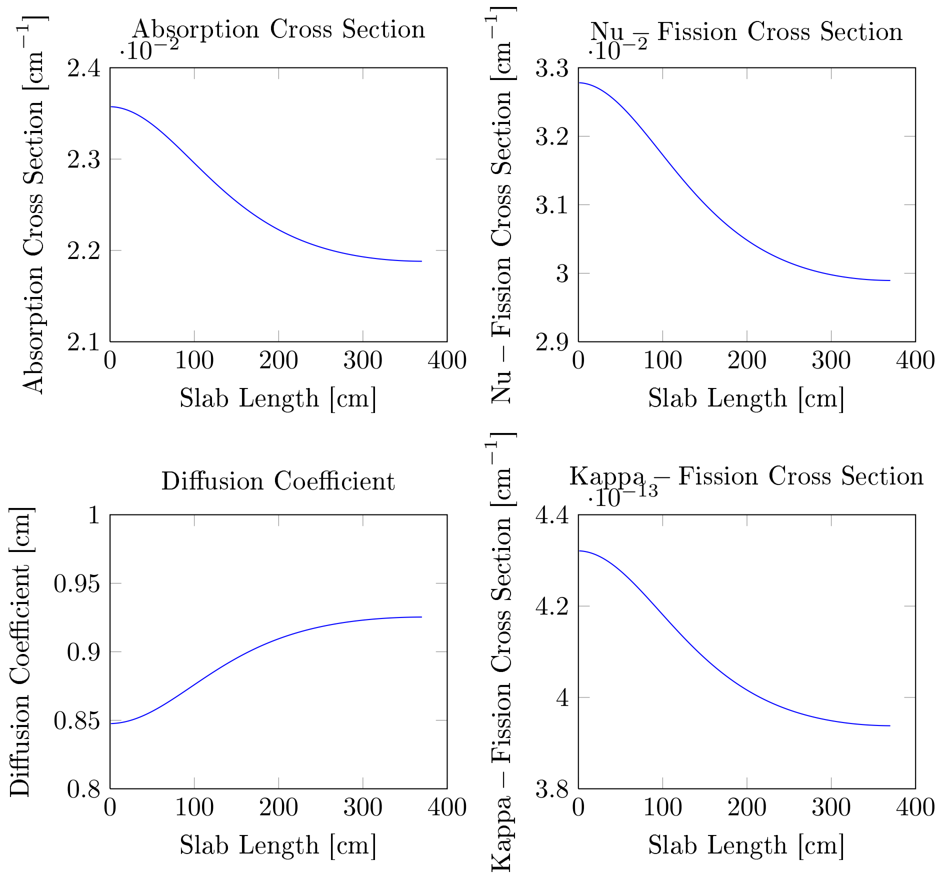


Figure 9.2. Results from Steady State Calculation Part 2

sections will be smaller. This proved to be effective for this situation and was used to compute the results presented in this section.

Finally, it is interesting to look at the distribution of how time was spent in the code. To do this, MATLAB's built in profiler was used. The results of the profiler are shown in Fig. 9.3. From the results, it is interesting to see that approximately 88% of the time was spent in the look-up X-Steam tables to relate density to temperature. Unfortunately, nothing can be done about this if the lookup tables are going to be used. An improvement on this would be to fit the density vs. temperature data with a polynomial in the range of interest and use that in the residual equations and Jacobian-vector product routines. A linear approximation was made for the density dependence on coolant temperature. The residual equation is of the form

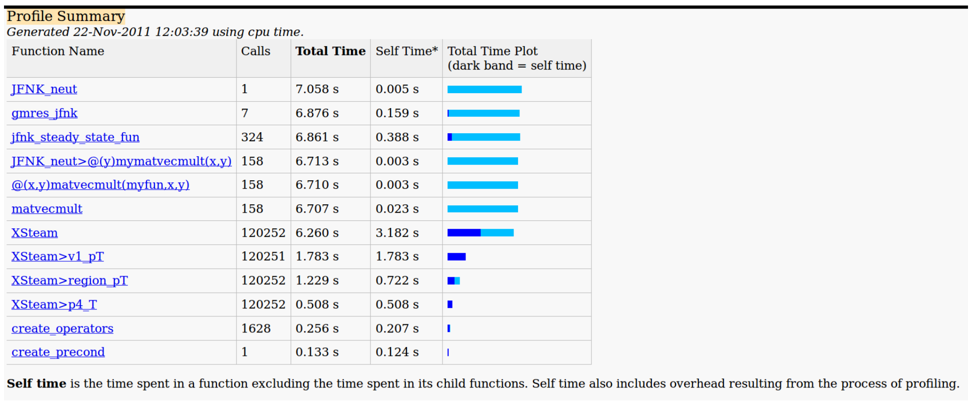
$$\mathcal{P} - \rho^{ref} - \frac{\partial \rho}{\partial T}\left(\mathbf{T} - T^{ref}\right), \tag{9.9}$$

where it was determined from the lookup tables that $\frac{\partial \rho}{\partial T} = -0.0023393\,\mathrm{g/cm^3/K}$. The computational cost was reduced significantly. The MATLAB profile summary is shown in Fig. 9.4. The total time to run the steady state calculation is now only 0.436 seconds as compared to 7 seconds with the look-up table. Therefore, this method will be used for the transient analysis. Source code for these routines is listed in Appendix B.3.

## 10 Transient Coupled Physics Solution

After the steady state solution is achieved, the transient simulation can be performed. Here, the normalization constant $\tilde{c}$ and eigenvalue $\lambda$ are now supplied as constants. For the transient calculations, a new set of residual equations was formulated. They are

$$\mathbf{F} = \left[\begin{array}{c} \mathbf{\Phi}^{n+1} - \mathbf{\Phi}^{n} + v\Delta t\left[\mathbb{M}\mathbf{\Phi}^{n+1} - (1-\beta)\,\lambda\mathbb{F}\mathbf{\Phi}^{n+1} - \lambda_d \mathbf{c}^{n+1}\right] \\ \mathbf{c}^{n+1} - \mathbf{c}^{n} + \Delta t\left(\lambda_d \mathbf{c}^{n+1} - \beta\lambda\mathbb{F}\mathbf{\Phi}^{n+1}\right) \\ \mathbf{Q} - \tilde{c}\mathbb{E}\mathbf{\Phi}\Delta x \\ \mathbf{T}^{n+1} - \mathbf{T}^{n} + \frac{w\Delta t}{\mathcal{P}^{n+1}A\Delta x}\left(\mathbb{S}\mathbf{T}^{n+1} - \mathbb{R}\mathbf{Q}^{n+1}\right) \\ \mathcal{P} - \rho^{ref} - \frac{\partial \rho}{\partial T}\left(\mathbf{T} - T^{ref}\right) \\ \mathbf{\Sigma}_a - \Sigma_a^{ref} - \frac{\partial \Sigma_a}{\partial \rho}\left[\mathcal{P} - \rho^{ref}\right] \\ \nu\mathbf{\Sigma}_f - \nu\Sigma_f^{ref} - \frac{\partial \nu\Sigma_f}{\partial \rho}\left[\mathcal{P} - \rho^{ref}\right] \\ \mathbf{D} - D^{ref} - \frac{\partial D}{\partial \rho}\left[\mathcal{P} - \rho^{ref}\right], \\ \kappa\mathbf{\Sigma}_f - \kappa\Sigma_f^{ref} - \frac{\partial \kappa\Sigma_f}{\partial \rho}\left[\mathcal{P} - \rho^{ref}\right] \end{array}\right], \tag{10.1}$$

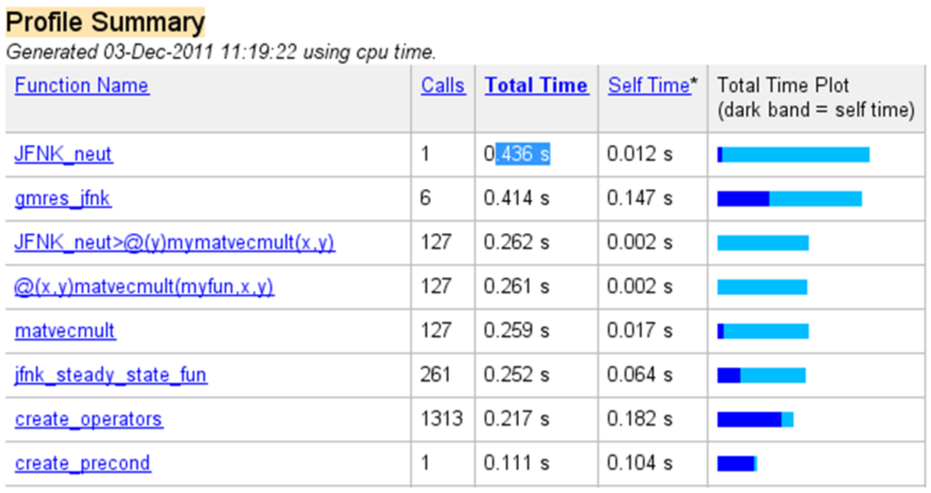

Profile Summary

*Generated 22-Nov-2011 12:03:39 using cpu time.*

| Function Name | Calls | **Total Time** | Self Time* | Total Time Plot (dark band = self time) |
|---|---|---|---|---|
| JFNK_neut | 1 | 7.058 s | 0.005 s | |
| gmres_jfnk | 7 | 6.876 s | 0.159 s | |
| jfnk_steady_state_fun | 324 | 6.861 s | 0.388 s | |
| JFNK_neut>@(y)mymatvecmult(x,y) | 158 | 6.713 s | 0.003 s | |
| @(x,y)matvecmult(myfun,x,y) | 158 | 6.710 s | 0.003 s | |
| matvecmult | 158 | 6.707 s | 0.023 s | |
| XSteam | 120252 | 6.260 s | 3.182 s | |
| XSteam>v1_pT | 120251 | 1.783 s | 1.783 s | |
| XSteam>region_pT | 120252 | 1.229 s | 0.722 s | |
| XSteam>p4_T | 120252 | 0.508 s | 0.508 s | |
| create_operators | 1628 | 0.256 s | 0.207 s | |
| create_precond | 1 | 0.133 s | 0.124 s | |

**Self time** is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 9.3. Profiler Results for steady state Case



**Profile Summary**

*Generated 03-Dec-2011 11:19:22 using cpu time.*

| Function Name | Calls | Total Time | Self Time* | Total Time Plot (dark band = self time) |
|---|---|---|---|---|
| JFNK_neut | 1 | 0.436 s | 0.012 s | |
| gmres_jfnk | 6 | 0.414 s | 0.147 s | |
| JFNK_neut>@(y)mymatvecmult(x,y) | 127 | 0.262 s | 0.002 s | |
| @(x,y)matvecmult(myfun,x,y) | 127 | 0.261 s | 0.002 s | |
| matvecmult | 127 | 0.259 s | 0.017 s | |
| jfnk_steady_state_fun | 261 | 0.252 s | 0.064 s | |
| create_operators | 1313 | 0.217 s | 0.182 s | |
| create_precond | 1 | 0.111 s | 0.104 s | |

Figure 9.4. MATLAB Profile Summary with Linear Fit for Density

with unknown vector

$$\mathbf{x}^{n+1} = \begin{bmatrix} \mathbf{\Phi} \\ \mathbf{c} \\ \mathbf{Q} \\ \mathbf{T} \\ \mathcal{P} \\ \mathbf{\Sigma}_a \\ \nu\mathbf{\Sigma}_f \\ \mathbf{D} \\ \kappa\mathbf{\Sigma}_f \end{bmatrix}. \tag{10.2}$$

A Jacobian matrix was constructed from the steady solution of all of the above unknowns for the purposes of creating a preconditioner. This preconditioner is only formed once for the whole transient analysis. The Jacobian is

$$\mathbb{J} = \left[\begin{array}{c|c|c|c|c|c|c|c|c}
\begin{matrix}\mathbb{I} + v\Delta t \times \\ \left[\mathbb{M} - (1-\beta)\lambda\mathbb{F}\right]\end{matrix} & -v\Delta t\lambda_d\mathbb{I} & 0 & 0 & 0 & v\Delta t \mathrm{diag}\left\{\Phi\right\} & \begin{matrix} v\Delta t\,(1-\beta) \times \\ -\lambda \mathrm{diag}\left\{\Phi\right\}\end{matrix} & v\Delta t\mathbb{MD} & 0 \\ \hline
-\Delta t\beta\lambda\mathbb{F} & \mathbb{I} + \Delta t\lambda_d\mathbb{I} & 0 & 0 & 0 & 0 & -\Delta t\beta\lambda\mathrm{diag}\left\{\Phi\right\} & 0 & 0 \\ \hline
-\mathbb{E}\tilde{c}\Delta x & 0 & \mathbb{I} & 0 & 0 & 0 & 0 & 0 & -\tilde{c}\Delta x \mathrm{diag}\left\{\Phi\right\} \\ \hline
0 & 0 & -\frac{w\Delta t}{\mathcal{P}A\Delta x}\mathbb{R} & \mathbb{I} + \frac{w\Delta t}{\mathcal{P}A\Delta x}\mathbb{S} & \begin{matrix}\mathrm{diag}\left\{-\frac{w\Delta t}{\mathcal{P}^2 A\Delta x} \times \right. \\ \left. (\mathbb{S}\mathbf{T} - \mathbb{R}\mathbf{Q})\right\}\end{matrix} & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & -\frac{\partial\rho}{\partial T}\mathbb{I} & \mathbb{I} & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & -\frac{\partial\Sigma_a}{\partial\rho} & \mathbb{I} & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & -\frac{\partial\nu\Sigma_f}{\partial\rho} & 0 & \mathbb{I} & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & -\frac{\partial D}{\partial\rho} & 0 & 0 & \mathbb{I} & 0 \\ \hline
0 & 0 & 0 & 0 & -\frac{\partial\kappa\Sigma_f}{\partial\rho} & 0 & 0 & 0 & \mathbb{I}
\end{array}\right]. \tag{10.3}$$

As in the steady state case, a zero-fill ILU preconditioner was formed from this Jacobian matrix. The transient that will be simulated is an insertion of a control rod followed by a withdrawal of that control rod. At every time step, the control rod insertion length is set corresponding to Fig. 10.1. Since the equations are being solved implicitly, the JFNK solver is used at every time step to solve for the unknown vector above. The behavior of the reactor is shown in Fig. 10.2 and core average temperature is shown in Fig. 10.3.

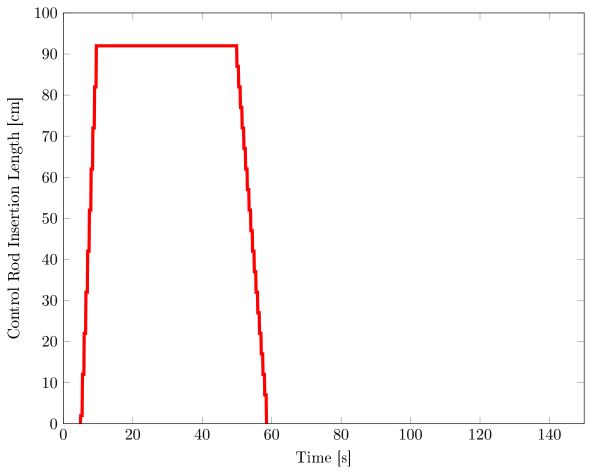


Figure 10.1. Control Rod Insertion Scheme

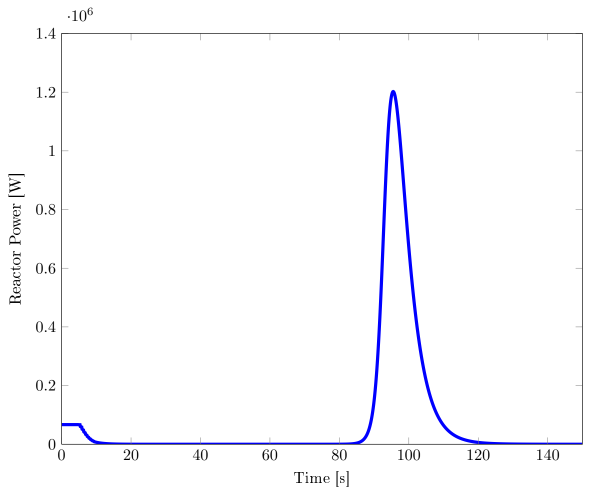


Figure 10.2. Power Behavior from Control Insertion/Withdrawal

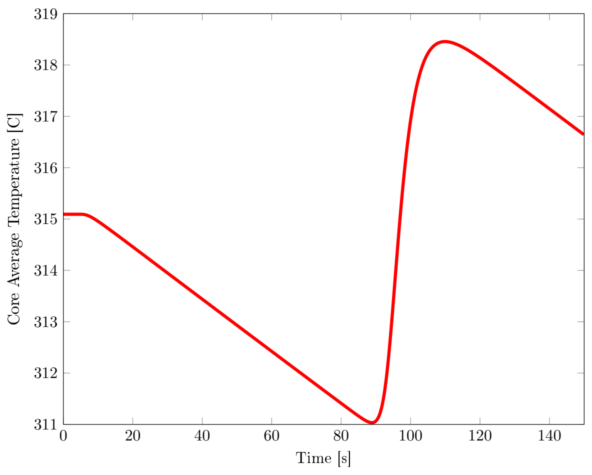


Figure 10.3. Core Average Temperature Behavior from Control Insertion/Withdrawal

When the control rod was inserted, the power immediately decreased, while the temperature gradually decreased. This delayed effect caused the power to decrease significantly before the control rod was withdrawn and thermal feedback started to increase power again. The power rose very quickly until the temperature of the coolant was large enough so that the rate of fission was decreased. The power then started to fall rapidly until the end of the simulation. The effects seen here are in an out of phase nature. The responses will continue to oscillate until a steady state is reached again. This type of response is not desired in a nuclear reactor due to heavy thermal stress that may occur. The main reason for this effect is that not all of the physics was modeled. In reality, the average temperature of the fuel will have a very strong prompt feedback. Thus, the feedback will occur on a faster time scale.

The response of the coolant in simulation is slow partly due to the mass flux and heat capacity of the coolant. To achieve a faster response, the mass flux was increased by reducing the flow area. This will also reduce the heat capacity. Therefore, the fluid will move faster through the medium and the feedback will occur more quickly. The flow area was reduced to $A = 0.867 \times 10^{-5}\,\text{cm}^2$. The same simulation was performed with this new flow area. The response of the power and core average temperature is shown in Figs. 10.4 and 10.5.

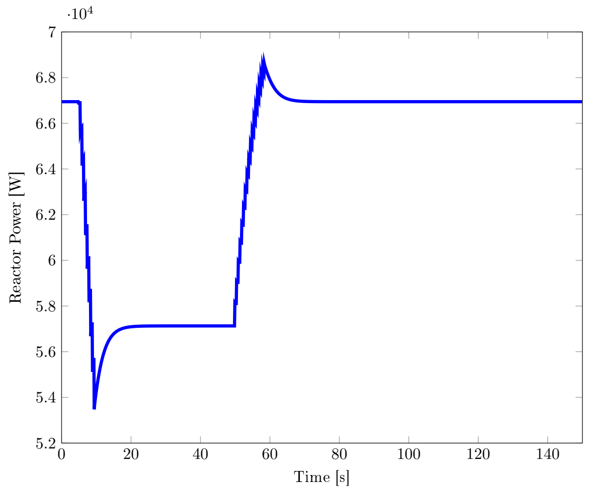


Figure 10.4. Power Behavior from Control Insertion/Withdrawal with High Mass Flux

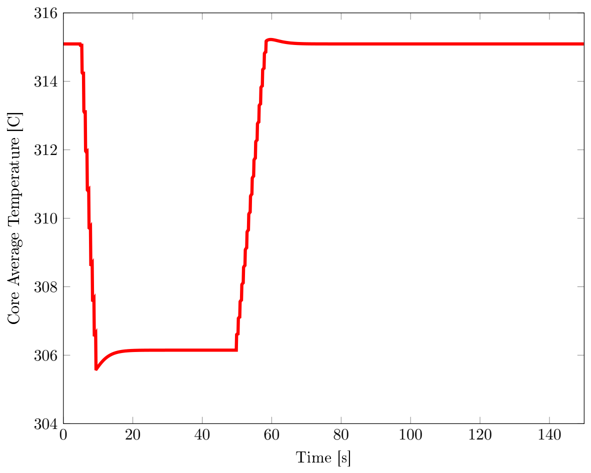


Figure 10.5. Core Average Temperature Behavior from Control Insertion/Withdrawal with High Mass Flux

In this simulation, the coolant density response follows the power response very closely. The power reaches a new steady state with the control rod inserted. This is due to the thermal feedback balancing the control rod insertion effect. When the control rod is removed, the power increases and then settles back at the original steady state. A comparison was also made for the flux shapes and density distribution for the steady state conditions when the rod is partially inserted and withdrawn. These comparisons are shown in Figs. 10.6 and 10.7.

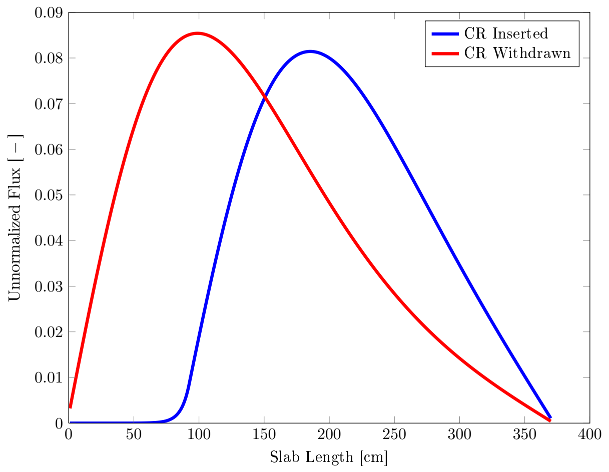


Figure 10.6. Steady State Flux Shape with and without Control Rod Inserted

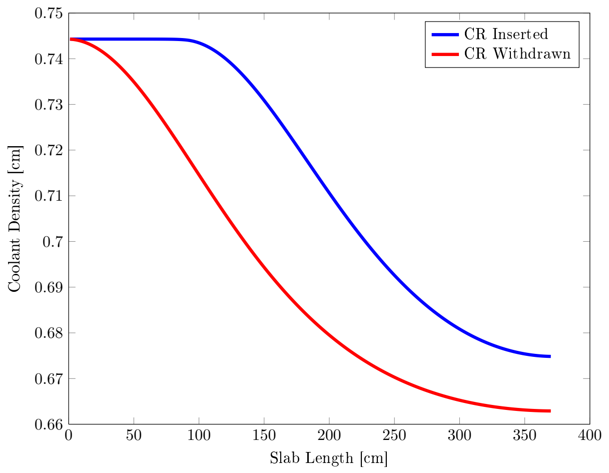


Figure 10.7. Steady State Density Distribution with and without Control Rod Inserted

It can be observed that when the control rod is inserted in the reactor, the flux is depressed in that region, since neutrons are being absorbed and not causing fission. The flux peak is also lowered and shifted to the right from where it was when the rod was not inserted. Since there is no power being generated in the leftmost part of the reactor, the density remains constant during this region. It can also be observed that the average density is higher when the rod is partially inserted. The results are consistent with the interaction physics for this transient. For these transient calculations, about 6-7 Newton iterations were required at each time step. For a given Newton iteration, about 20-30 GMRES inner iterations were required to converge the linear step. Source code for these routines is listed in Appendix B.4. An animation of this transient is shown in Fig. 10.8.

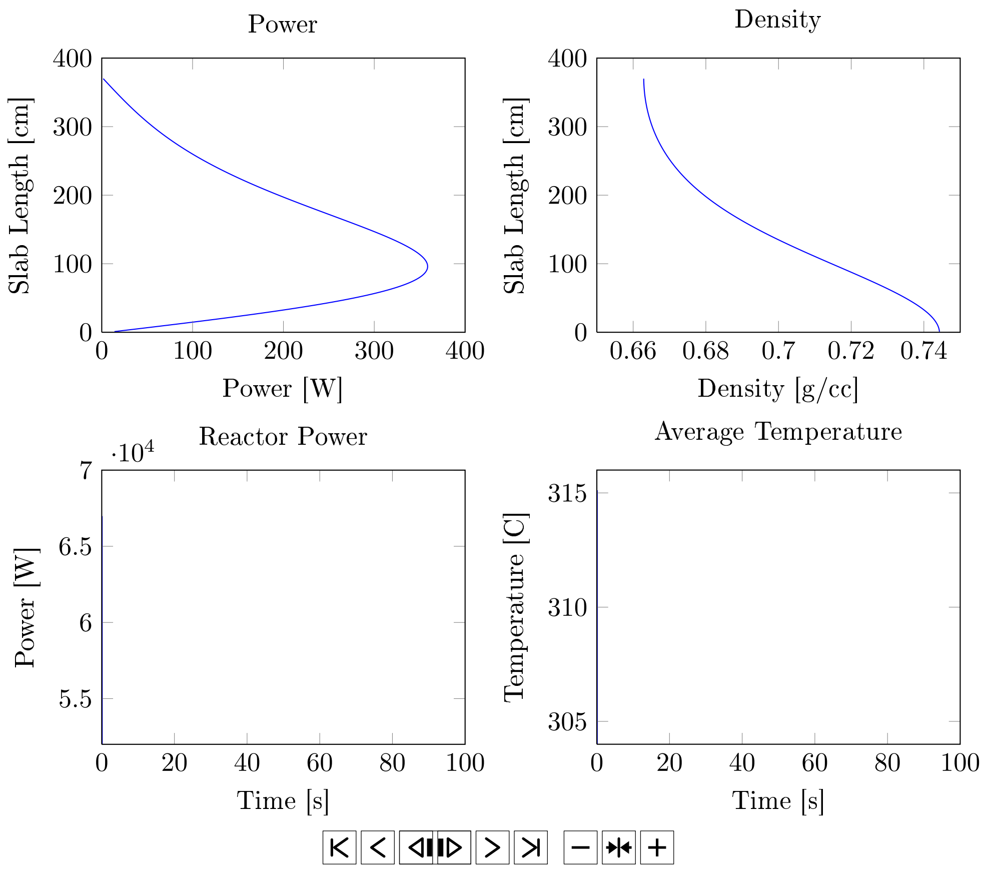


Figure 10.8. Transient Animation

# 11 Conclusions and Future Work

In this work, a Jacobian-Free Newton-Krylov framework was established and tested for both eigenvalue and nonlinear coupled physics problems. The solvers seem to be robust as long as the Jacobian-vector approximation is scaled appropriately. The only time the solvers broke down was when the original implementation of fission energy was employed. Since the magnitude of the neutron flux normalization parameter is so large and the energy per fission is so small, i.e. 26 orders of magnitude between the values, scaling problems arose. These scaling problems were fixed by altering the magnitude of energy produced from fission such that the normalization constant is of similar magnitude.

It was also observed that since the neutronics steady state calculation is an eigenvalue problem, any eigenpair can satisfy the nonlinear equations. This problem of converging on a different mode was observed in reactors with a high dominance ratio. To circumvent this problem, a few number of power iterations were run to get a rough fundamental mode flux shape to assist in the convergence to this mode in the nonlinear solver. For each nonlinear calculation a preconditioner was necessary to limit the number of iterations in the GMRES solver. This preconditioner was only constructed once at the beginning of the nonlinear iteration loop. For the steady state calculations, the result from a few power iterations was used to construct the Jacobian, whereas in the transient case, the steady results were used to form the preconditioner.

It was observed that the majority of the time was spent looking up thermodynamic properties from X-Steam. This is unacceptable and a better method must be employed in the future. One solution is to create separate lookup tables in the region of interest from X-Steam and use this new routine in the code. In this routine a faster search algorithm can be implement to ensure this is not the bottleneck of the calculation. Another solution is to fit the dependence with a high order polynomial. In the transient calculation, only a linear dependence was used to just get more realistic execution times.

For future work, this code needs to be rewritten in a compiled language such as Fortran. Instead of using the manual solvers created in MATLAB, the PETSc library will be used for the Newton iteration and GMRES solution. This will also give a better indication on the computational cost of performing these calculations. An immediate improvement would be to implement a heat conduction model so that fuel temperature feedback can be modeled. This feedback is more important than the coolant density feedback used in this work. This will hopefully give more of the prompt behavior that is expected in these transients. Finally, higher order time integration schemes could be implemented instead of the simple first order implicit Euler. These could include predictor-corrector, Runge-Kutta or multistep methods.

# References

[1] George Bell and Samuel Glasstone. *Nuclear Reactor Theory.* Van Nostrand Reinhold Co., New York, 1970.

[2] T. Downar, D. Lee, Y. Xu, and V. Seker. *PARCS v3.0: U.S. NRC Core Neutronics Simulator. User Manual.* University of Michigan, 2009.

[3] James J. Duderstadt and Louis J. Hamilton. *Nuclear Reactor Analysis.* Wiley, New York, 1976.

[4] Daniel F. Gill. *Newton-Krylov Methods for the Solution of the k-Eigenvalue Problem in Multigroup Neutronics Calculations.* PhD thesis, The Pennsylvania State University, 2009.

[5] Alain Hébert. *Applied Reactor Physics.* Presses Internationales Polytechnique, Montréal, 2009.

[6] Magnus Holmgren. *X STEAM for MATLAB.* Excel Engineering, 2006.

[7] C.T. Kelly. *Iterative Methods for Linear and Nonlinear Equations.* Society for Industrial and Applied Mathematics, Philadelphia, 1987.

[8] C.T. Kelly. *Solving Nonlinear Equations with Newton's Method.* Society for Industrial and Applied Mathematics, Philadelphia, 2003.

[9] D.A. Knoll and D.E. Keyes. Jacobian-free Newton-Krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193:357–397, 2004.

[10] Jaakko Leppänen. *PSG2/Serpent: A Continuous-energy Monte Carlo Reactor Physics Burnup Code User's Manual.* VTT, Espoo, December 2010.

[11] Planet Math. Given rotation. *http://planetmath.org/encyclopedia/GivensRotation.html*, 2011.

[12] V.A. Mousseau. Implicitly balanced solution of the two-phase flow equations couple to nonlinear heat conduction. *Journal of Computational Physics*, 200:104–132, 2004.

[13] RELAP5-3D. *RELAP5-3D Code Manual Volume II: User's Guide and Input Requirements.* Idaho National Laboratory, 2.4 edition, June 2005.

[14] Youcef Saad and Martin H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *Society for Industrial and Applied Mathematics*, 7:856–869, 1986.

[15] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Societry for Industrial and Applied Mathematics, 2003.

[16] Neil E. Todreas and Mujid S. Kazimi. *Nuclear Systems I.* Taylor & Francis, New York, second edition, 2011.

[17] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra.* Societry for Industrial and Applied Mathematics, 1997.

[18] Eugene L. Wachspress. *Iterative Solution of Elliptic Systems and Applications to the Neutron Diffusion Equations of Reactor Physics.* PRENTICE-HALL, Englewood Cliffs, 1966.

# A Source Code for General Solvers

## A.1 Inexact Newton Nonlinear Solver

```
function x = inexact_newton(myfun,mymatvecmult,xo,L,U)

% set guess to x vector
x = xo;
dx = 0.01*ones(length(xo),1);

% begin Newton iteration
for i = 1:10000

    % evaluate function
    F = myfun(x);

    % compute residual norm
    res = norm(F);

    % Display residual
    fprintf('Iter: %d     Res: %d',i,res);

    % exit if residual is low
    if res < 1e-6
        fprintf('\n');
        break
    end

    % construct matrix vector multiply @ x
    mymatvecmult_x = @(y) mymatvecmult(x,y);

    % GMRES solution
    [dx,err] = gmres_jfnk(L,U,-F,dx,mymatvecmult_x,1000,5,1e-8,1e-10);

    % check GMRES convergence
    if find(err,1,'last') == length(err)
        error('GMRES solver did not converge')
    end

    % print iteration number
    fprintf('  GMRES iters: %d  GMRES res: %d\n',find(err,1,'last'),err(find(err,1,'last')));

    % update guess
    x = x + dx;

end
```

## A.2 GMRES Linear Solver w/ Preconditioning and Givens Rotation

```
function [x,err] = gmres_jfnk(L,U,b,x,matvec,res,maxiter,tol,mtol)

% get size of b
m = length(b);

% preallocate for max storage
Q = zeros(m,res+1);
H = zeros(res+1,res);
err = zeros(maxiter*res,1);
c = zeros(res+1,1);
s = zeros(res+1,1);

% re-set tolerance
```

```
scale = norm(U\(L\b));
tol = tol*scale;
tol = max(tol,mtol);

% begin while loop
for k = 1:maxiter

    % compute A*x with function
    Ax = matvec(x);

    % compute initial residual
    r = U\(L\(b - Ax));
    beta = norm(r);

    % set up g vector
    g = beta*eye(res+1,1);

    % check convergence
    if beta < tol
        break
    end

    % compute first q
    Q(:,1) = r/beta;

    % begin iteration
    for n = 1:res

        % begin Arnoldi for step n
        % compute vector v
        y = Q(:,n);
        Ay = matvec(y);
        v = U\(L\(Ay));
        normv1 = norm(v);

        % loop around all previous vectors
        for j = 1:n

            % compute orthogonal projection of A onto new Krylov subspace
            % H = Q'*A*Q
            H(j,n) = Q(:,j)'*v;

            % solves equation 33.4 in Trefethen so that Aq = h_(n+1)q_(n+1)
            v = v - H(j,n)*Q(:,j);

        end

        % compute new h
        H(n+1,n) = norm(v);
        normv2 = H(n+1,n);

        % Reorthogonalize
        if (normv1 + 0.001*normv2 == normv1)
            for j = 1:n
                htmp = v(:,j)'*v;
                H(j,n) = H(j,n) + htmp;
                v = v - htmp*Q(:,j);
            end
            H(n+1,n) = norm(v);
        end

        % solve for next column
        Q(:,n+1) = v/H(n+1,n);

        % apply givens rotation
        [H,c,s,g] = given_rot(H,c,s,g,n);

        % get error
        err(n+res*(k-1)) = abs(g(n+1));

        % check convergence
        if err(n+res*(k-1)) < tol
            break
        end

    end

    % compute y
    y = H(1:n,1:n)\g(1:n);

    % compute x
    x = x + Q(:,1:n)*y;

    % check convergence
    if err(k*n) < tol
        %err = err/scale;
        break

    end

end

end

function [H,c,s,g] = given_rot(H,c,s,g,n)

        % apply all previous and current rotations to new column
        for i = 1:n-1

            % apply rotation and store in temporary vars
            tmp1 = c(i)*H(i,n) - s(i)*H(i+1,n);
            tmp2 = s(i)*H(i,n) + c(i)*H(i+1,n);

            % computation complete move temp vars to matrix
            H(i,n) = tmp1;
            H(i+1,n) = tmp2;

        end

        % compute new c and s
        c(n) = H(n,n)/sqrt(H(n,n)^2+H(n+1,n)^2);
        s(n) = -H(n+1,n)/sqrt(H(n,n)^2+H(n+1,n)^2);

        % apply rotation and store in temporary vars for step n
        tmp1 = c(n)*H(n,n) - s(n)*H(n+1,n);
        tmp2 = s(n)*H(n,n) + c(n)*H(+1,n);

        % computation complete move temp vars to matrix
        H(n,n) = tmp1;
        H(n+1,n) = tmp2;

        % zero out i+1 element
        H(n+1,n) = 0.0;

        % apply givens rotation to right hand side
        tmp1 = c(n)*g(n) - s(n)*g(n+1);
        tmp2 = s(n)*g(n) + c(n)*g(n+1);
        g(n) = tmp1;
        g(n+1) = tmp2;

end
```

## A.3 Power Iteration

```
function [keignew,phinew] = power_iter(M,F,phi,keig,iters)

% begin power iteration loop
for iter = 1:iters

    % Update Flux
    phinew = M\(1/keig*F*phi);

    % Update Keff
    keignew = keig*sum((F*phinew).*(F*phinew))/sum((F*phi).*(F*phinew));

    % Calculate Difference
    ferr = (norm(phinew-phi));
    kerr = abs(keignew-keig)/keignew;

    % Display output
    fprintf('Iter: %d     Err: %d\n',iter,ferr);

    % Check Convergence
    if ferr < 1.0e-10 && kerr < 1.0e-10
        break
    else
        phi = phinew;
        keig = keignew;
    end

end
```

### A.4 Finite Difference Jacobian-vector Multiplication

```
function Ay = matvecmult(myfun,x,y)

% set parameters
b = 1e-8;
N = length(y);

% compute epsilon
epsilon = b*sum(x)/(N*norm(y));

% approximate matrix vector multiplication
Ay = (myfun(x+epsilon*y) - myfun(x))/epsilon;

end
```

## B Source Code for Calculations

### B.1 Input File and Main Code

```
% Bryan Herman
% JFNK input file
% 2.29 Numerical Fluid Mechanics

global info geom th neut

%% Control Information

% maximum time
info.time = 100.0;
```

```
% time step
info.dt = 0.1;

%% geometry object

% number of mesh cells
geom.n = 370;

% dimension of mesh [cm]
geom.dx = 1;

%% thermal hydraulic object

% Inlet Temperature [C]
th.Tin = 293.1;

% Mass Flow Rate in Subchannel [kg/s]
th.w = 0.335;

% Interior Subchannel flow area [cm^2]
th.area = 0.879e-5;

% Average power per subchannel [W]
th.Qr = 3411e6/(193*264);

% Average temperature [C]
th.Tref = 310;

% System pressure [bar]
th.P = 155;

% Reference coolant density [g/cc] @ 155 bar and 310 C
th.rhoREF = XSteam('rho_pT',th.P,th.Tref)/1000;

% Specific heat
th.cp = XSteam('cp_pT',th.P,th.Tref)*1000;

% Slope
th.DrhoDtemp = -0.0023393;

%% neutronic object

% set boundary conditions
neut.alb = [0,0];

% set delayed neutron information
neut.beta = 0.00682;
neut.lambd = 0.815922;
neut.vel = 3.2047e+006;

% set reference neutronic parameters
neut.absxsREF = 2.27516E-02;
neut.nfissREF = 3.13791E-02;
neut.diffREF = 8.85342E-01;
neut.kfissconst = 1e12;
neut.kfissREF = 4.13494E-13*neut.kfissconst;

% dependence of neutronic parameters on density
neut.skew = 1;
neut.DabsxsDrho = 0.020796*neut.skew;
neut.DnfissDrho = 0.035471*neut.skew;
neut.DdiffDrho  = -0.95551*neut.skew;
neut.DkfissDrho = 4.7055E-13*neut.kfissconst*neut.skew;
```

```
% Bryan Herman
% 2.29 Numerical Fluid Mechanics Term Project
% Steady State couple physics
clear; close all; clear -global
path(path,'../../Steam')

% input file
jfnk_input

% choose execution option (1-Neutronics Static, 2-Coupled Static,
% 3-Transient)
opt = 2;

% turn on profiler
%profile on -timer cpu

% check for neutronics only
if opt == 1
    myfun = @neut_res_fun;
    x_neut = run_neut_only(myfun);
end

% check for coupled static or transient
if (opt == 2 || opt == 3)
    myfun = @coupled_steady_res_fun;
    x_steady = run_coupled_static(myfun);
end

% check for control rod transient
if opt == 3
   myfun = @coupled_trans_res_fun;
   mycr = @control_rod;
   [x_trans,pow,tave,rod] = run_coupled_trans(myfun,mycr,x_steady);
end

% terminate profiler
%profile viewer
%profile off
%profsave(profile('info'),'prof_results')
```

## B.2 Steady State Neutronics only

```
function x = run_neut_only(myfun)

global geom neut

% get problem size
n = geom.n;

% build loss matrix
M = create_operators_neut('M',geom,neut);

% build production matrix
F = create_operators_neut('F',geom,neut);

% set initial guess for power iteration
phi = ones(n,1);
keff = 1.0;

% get seeded with initial power iterations
[keff,phi] = power_iter(M,F,phi,keff,2);

% construct preconditioner
[L,U] = create_precond_neut(M,F,phi,1/keff);

% set up initial guess for calculation
```

```
x = [phi;1/keff];

% steady state function evaluation (set operators)
myfun_eval = @(xx) myfun(M,F,xx);

% set up Jacobian-vector multiplication routine
myJacvecmult = @(x,y) Jacobian_vec_FD(myfun_eval,x,y);

% run static calculation
x = inexact_newton(myfun_eval,myJacvecmult,x,L,U);

end
```

```
function out_mat = create_operators_neut(oper,geom,neut)

% extract problem size
n = geom.n;

% Find Operator to Create
switch oper

    % Create M operator
    case 'M'

        % number of nonzeros
        nnz = n + 2*(n-1);

        % preallocate sparse matrix vectors
        rowvec = zeros(nnz,1);
        colvec = zeros(nnz,1);
        valvec = zeros(nnz,1);

        % extract data from data objects
        alb = neut.alb;
        dx = geom.dx;

        % extract needed data from x vector
        D = neut.diffREF*ones(n,1);
        absxs = neut.absxsREF*ones(n,1);

        % shift diffusion coefficient vector
        Dm1 = circshift(D,1);
        Dp1 = circshift(D,-1);

        % create diagonal (left boundary, interior, right boundary)
        rowvec(1:n) = 1:n;
        colvec(1:n) = 1:n;

        % left boundary
        valvec(1) = 2/dx*((D(1)*(1-alb(1)))/(4*D(1)*(1+alb(1)) + dx*(1-alb(1))) + ...
            1/dx*(D(1)*D(2))/(D(1) + D(2))) + absxs(1);

        % interior cells
        valvec(2:n-1) = 2/dx^2*((Dm1(2:n-1).*D(2:n-1))./(Dm1(2:n-1) + D(2:n-1)) + ...
            (D(2:n-1).*Dp1(2:n-1))./(D(2:n-1) + Dp1(2:n-1))) + absxs(2:n-1);

        % right boundary
        valvec(n) = 2/dx*(1/dx*(D(n-1)*D(n))/(D(n-1) + D(n)) + (D(n)*(1-alb(2))) / ...
            (4*D(n)*(1+alb(2)) + dx*(1-alb(2)))) + absxs(n);

        % create subdiagonal -1
        rowvec(n+1:2*n-1) = 2:n;
        colvec(n+1:2*n-1) = 1:n-1;
        valvec(n+1:2*n-1) = -2/dx^2*((Dm1(2:n).*D(2:n))./(Dm1(2:n) + D(2:n)));
        % create superdiagonal +1
        rowvec(2*n:3*n-2) = 1:n-1;
        colvec(2*n:3*n-2) = 2:n;
        valvec(2*n:3*n-2) = -2/dx^2*((D(1:n-1).*Dp1(1:n-1))./(D(1:n-1) +Dp1(1:n-1)));

        % create sparse matrix for output
        out_mat = sparse(rowvec,colvec,valvec);

    % Create F operator
    case 'F'

        % number of nonzeros
        nnz = n;

        % extract nfiss from x vector
        nfiss = neut.nfissREF*ones(n,1);

        % preallocate sparse matrix vectors
        rowvec = zeros(nnz,1);
        colvec = zeros(nnz,1);
        valvec = zeros(nnz,1);

        % calculate diagonal
        rowvec(1:n) = 1:n;
        colvec(1:n) = 1:n;
        valvec(1:n) = nfiss(1:n);

        % create sparse matrix for output
        out_mat = sparse(rowvec,colvec,valvec);

    case default

        error('Operator does not exist!');

end

end
```

```
function [L,U] = create_precond_neut(M,F,phi,lamb)

% get problem size
n = size(M,1);

% number of nonzeros
nnz = n + 2*(n-1) + 2*n + 1;

% preallocate Jacobian
J = spalloc(n+1,n+1,nnz);

% place terms in J
J(1:n,1:n) = M - lamb*F;
J(1:n,n+1) = -F*phi;
J(n+1,1:n) = -phi';

% place approx term for eigenvalue
J(n+1,n+1) = 1;

% ilu-0 preconditioner
setup.type = 'nofill';
[L,U] = ilu(J,setup);

end
```

```
function Fun = neut_res_fun(M,F,x)

% get problem size
n = size(M,1);

% preallocate function eval
Fun = zeros(n+1,1);

% extract data
phi = x(1:n);
lamb = x(n+1);

% evaluate residual
Fun(1:n) = M*phi - lamb*F*phi;
Fun(n+1) = -1/2*(phi')*phi + 1/2;

end
```

### B.2.1 Analytic Jacobian-vector Product Routine

```
function Jy = Jacobian_vec_mult(M,F,x,y)

% get problem size
n = size(M,1);

% preallocate Jy vector
Jy = zeros(n+1,1);

% extract info
phi = x(1:n);
lamb = x(n+1);

% perform Jy multiplication
Jy(1:n) = (M - lamb*F)*y(1:n) + (-F*phi)*y(n+1);
Jy(n+1) = (-phi')*y(1:n);

end
```

## B.3 Steady State Coupled Neutronics/Thermal Hydraulics

```
function x = run_coupled_static(myfun)

% get initial guess
x = get_initial_guess();

% build preconditioner
[L,U] = create_precond_steady(x);

% set up Jacobian-vector multiplication routine
myJacvecmult = @(x,y) Jacobian_vec_FD(myfun,x,y);

% run steady state
x = inexact_newton(myfun,myJacvecmult,x,L,U);

% plot results
create_plots_static(x);

end
```

```
function x = get_initial_guess()

global geom neut th

% get problem size
n = geom.n;
dx = geom.dx;

% get constants from object
Qr = th.Qr; % reactor power
P = th.P; % pressure of system
Tin = th.Tin; % inlet temperature
w = th.w;
cp = th.cp;
absxsREF = neut.absxsREF;
nfissREF = neut.nfissREF;
diffREF = neut.diffREF;
kfissREF = neut.kfissREF;
DabsxsDrho = neut.DabsxsDrho;
DnfissDrho = neut.DnfissDrho;
DdiffDrho = neut.DdiffDrho;
DkfissDrho = neut.DkfissDrho;
rhoREF = th.rhoREF;

% form neutronics guess
x = ones(8*n+2,1);
x(1:n) = x(1:n)/norm(x(1:n));
x(4*n+2:5*n+1) = absxsREF*ones(n,1);
x(5*n+2:6*n+1) = nfissREF*ones(n,1);
x(6*n+2:7*n+1) = diffREF*ones(n,1);

% evaluate neutronic operators
M = create_operators_steady('M',x);
F = create_operators_steady('F',x);

% perform one power iteration
[keff,x(1:n)] = power_iter(M,F,x(1:n),1/x(8*n+2),20);
x(8*n+2) = 1/keff;

% renormalize phi
x(1:n) = x(1:n)/norm(x(1:n));

% compute normalization parameter
x(n+1) = Qr/(dx*(kfissREF*ones(n,1))'*x(1:n));

% compute individual energy deposition
x(n+2:2*n+1) = x(n+1)*kfissREF*dx*x(1:n);

% compute temperature and density distribution
for i = 1:n

    % calculate temperature (watch initial condition)
    if i == 1
        x(2*n+2) = Tin + x(n+2)/(2*w*cp);
    else
        x(2*n+1+i) = x(2*n+1+(i-1)) + (x(n+1+i) + x(n+1+(i-1)))/(2*w*cp);
    end

    % calculate density at pressure and computed temperature
    x(3*n+1+i) = XSteam('rho_pT',P,x(2*n+1+i))/1000;

end

% compute new cross sections
x(4*n+2:5*n+1) = absxsREF + DabsxsDrho*(x(3*n+2:4*n+1) - rhoREF);
x(5*n+2:6*n+1) = nfissREF + DnfissDrho*(x(3*n+2:4*n+1) - rhoREF);
x(6*n+2:7*n+1) = diffREF + DdiffDrho*(x(3*n+2:4*n+1) - rhoREF);
```

```
x(7*n+2:8*n+1) = kfissREF + DkfissDrho*(x(3*n+2:4*n+1) - rhoREF);

end
```

```
function out_mat = create_operators_steady(oper,x)

global geom neut th

% extract problem size
n = geom.n;

% Find Operator to Create
switch oper

    % Create M operator
    case 'M'

        % number of nonzeros
        nnz = n + 2*(n-1);

        % preallocate sparse matrix vectors
        rowvec = zeros(nnz,1);
        colvec = zeros(nnz,1);
        valvec = zeros(nnz,1);

        % extract data from data objects
        alb = neut.alb;
        dx = geom.dx;

        % extract needed data from x vector
        D = x(6*n+2:7*n+1);
        absxs = x(4*n+2:5*n+1);

        % shift diffusion coefficient vector
        Dm1 = circshift(D,1);
        Dp1 = circshift(D,-1);

        % create diagonal (left boundary, interior, right boundary)
        rowvec(1:n) = 1:n;
        colvec(1:n) = 1:n;

        % left boundary
        valvec(1) = 2/dx*((D(1)*(1-alb(1)))/(4*D(1)*(1+alb(1)) + dx*(1-alb(1))) + ...
            1/dx*(D(1)*D(2))/(D(1) + D(2))) + absxs(1);

        % interior cells
        valvec(2:n-1) = 2/dx^2*((Dm1(2:n-1).*D(2:n-1))./(Dm1(2:n-1) + D(2:n-1)) + ...
            (D(2:n-1).*Dp1(2:n-1))./(D(2:n-1) + Dp1(2:n-1))) + absxs(2:n-1);

        % right boundary
        valvec(n) = 2/dx*(1/dx*(D(n-1)*D(n))/(D(n-1) + D(n)) + (D(n)*(1-alb(2))) / ...
            (4*D(n)*(1+alb(2)) + dx*(1-alb(2)))) + absxs(n);

        % create subdiagonal -1
        rowvec(n+1:2*n-1) = 2:n;
        colvec(n+1:2*n-1) = 1:n-1;
        valvec(n+1:2*n-1) = -2/dx^2*((Dm1(2:n).*D(2:n))./(Dm1(2:n) + D(2:n)));

        % create superdiagonal +1
        rowvec(2*n:3*n-2) = 1:n-1;
        colvec(2*n:3*n-2) = 2:n;
        valvec(2*n:3*n-2) = -2/dx^2*((D(1:n-1).*Dp1(1:n-1))./(D(1:n-1) +Dp1(1:n-1)));

        % create sparse matrix for output
        out_mat = sparse(rowvec,colvec,valvec);
    % Create F operator
    case 'F'

        % number of nonzeros
        nnz = n;

        % extract nfiss from x vector
        nfiss = x(5*n+2:6*n+1);

        % preallocate sparse matrix vectors
        rowvec = zeros(nnz,1);
        colvec = zeros(nnz,1);
        valvec = zeros(nnz,1);

        % calculate diagonal
        rowvec(1:n) = 1:n;
        colvec(1:n) = 1:n;
        valvec(1:n) = nfiss(1:n);

        % create sparse matrix for output
        out_mat = sparse(rowvec,colvec,valvec);

    % Create E operator
    case 'E'

        % number of nonzeros
        nnz = n;

        % extract kfiss from x vector
        kfiss = x(7*n+2:8*n+1);

        % preallocate sparse matrix vectors
        rowvec = zeros(nnz,1);
        colvec = zeros(nnz,1);
        valvec = zeros(nnz,1);

        % calculate diagonal
        rowvec(1:n) = 1:n;
        colvec(1:n) = 1:n;
        valvec(1:n) = kfiss(1:n);

        % create sparse matrix for output
        out_mat = sparse(rowvec,colvec,valvec);

    % Create S operator
    case 'S'

        % number of nonzeros
        nnz = n + (n-1);

        % preallocate sparse matrix vectors
        rowvec = zeros(nnz,1);
        colvec = zeros(nnz,1);
        valvec = zeros(nnz,1);

        % calculate diagonal
        rowvec(1:n) = 1:n;
        colvec(1:n) = 1:n;
        valvec(1:n) = 1;

        % calculate subdiagonal -1
        rowvec(n+1:2*n-1) = 2:n;
        colvec(n+1:2*n-1) = 1:n-1;
        valvec(n+1:2*n-1) = -1;

        % create sparse matrix for output
        out_mat = sparse(rowvec,colvec,valvec);
```

```
% Create R operator
case 'R'

    % number of nonzeros
    nnz = n + (n-1);

    % preallocate sparse matrix vectors
    rowvec = zeros(nnz,1);
    colvec = zeros(nnz,1);
    valvec = zeros(nnz,1);

    % get data from object
    w = th.w;
    cp = th.cp;

    % calculate diagonal
    rowvec(1:n) = 1:n;
    colvec(1:n) = 1:n;
    valvec(1:n) = 1/(2*w*cp);

    % calculate subdiagonal -1
    rowvec(n+1:2*n-1) = 2:n;
    colvec(n+1:2*n-1) = 1:n-1;
    valvec(n+1:2*n-1) = 1/(2*w*cp);

    % create sparse matrix for output
    out_mat = sparse(rowvec,colvec,valvec);

% Create MD operator
case 'MD'

    % number of nonzeros
    nnz = n + 2*(n-1);

    % preallocate sparse matrix vectors
    rowvec = zeros(nnz,1);
    colvec = zeros(nnz,1);
    valvec = zeros(nnz,1);

    % extract data from data objects
    alb = neut.alb;
    dx = geom.dx;

    % extract needed data from x vector
    D = x(6*n+2:7*n+1);
    phi = x(1:n);

    % shift diffusion coefficient and phi vectors
    Dm1 = circshift(D,1);
    Dp1 = circshift(D,-1);
    phim1 = circshift(phi,1);
    phip1 = circshift(phi,-1);

    % create diagonal (left boundary, interior, right boundary)
    rowvec(1:n) = 1:n;
    colvec(1:n) = 1:n;

    % left boundary
    valvec(1) = 2/dx*((dx*(1-alb(1))^2)/(4*D(1)*(1+alb(1)) + dx*(1-alb(1)))^2 + ...
        1/dx*D(2)^2/(D(1) + D(2))^2)*phi(1);

    % interior cells
    valvec(2:n-1) = 2/dx^2*(Dm1(2:n-1).^2./(Dm1(2:n-1) + D(2:n-1)).^2 + ...
        Dp1(2:n-1).^2./(D(2:n-1) + Dp1(2:n-1)).^2).*phi(2:n-1);

    % right boundary
    valvec(n) = 2/dx*((dx*(1-alb(2))^2)/(4*D(n)*(1+alb(2)) + dx*(1-alb(2)))^2 + ...
            1/dx*D(n-1)^2/(D(n-1) + D(n))^2)*phi(n);

        % create subdiagonal -1
        rowvec(n+1:2*n-1) = 2:n;
        colvec(n+1:2*n-1) = 1:n-1;
        valvec(n+1:2*n-1) = -2/dx^2*(Dm1(2:n).^2./(Dm1(2:n) + D(2:n)).^2).*phim1(2:n);

        % create superdiagonal +1
        rowvec(2*n:3*n-2) = 1:n-1;
        colvec(2*n:3*n-2) = 2:n;
        valvec(2*n:3*n-2) = -2/dx^2*(Dp1(1:n-1).^2./(D(1:n-1) + Dp1(1:n-1)).^2).*phip1(1:n-1);

        % create sparse matrix for output
        out_mat = sparse(rowvec,colvec,valvec);

    case default

        error('Operator does not exist!');

end

end
```

```
function [L,U] = create_precond_steady(x)

global geom neut

% construct operators
M = create_operators_steady('M',x);
F = create_operators_steady('F',x);
E = create_operators_steady('E',x);
S = create_operators_steady('S',x);
R = create_operators_steady('R',x);
MD = create_operators_steady('MD',x);

% get number of meshes
n = geom.n;
dx = geom.dx;

% allocate function%
J = sparse(zeros(8*n+2)); % jacobian of function

% extract vectors from x
DabsxsDrho = neut.DabsxsDrho;
DnfissDrho = neut.DnfissDrho;
DdiffDrho = neut.DdiffDrho;
DkfissDrho = neut.DkfissDrho;

% extract vectors from x
phi = x(1:n);                   % fluxes
c_tilde = x(n+1);               % normalization constant
kfiss = x(7*n+2:8*n+1);         % kappa-fission macro xs
lamb = x(8*n+2);                % eigenvalue

% flux equation by flux
J(1:n,1:n) = (M - lamb*F);

% flux equation by absxs
J(1:n,4*n+2:5*n+1) = diag(phi);

% flux equation by nfiss
J(1:n,5*n+2:6*n+1) = -lamb*diag(phi);

% flux equation by diff
J(1:n,6*n+2:7*n+1) = MD;
```

```
% flux by eigenvalue
J(1:n,8*n+2) = -F*phi;

% norm by flux
J(n+1,1:n) = -c_tilde*dx*(kfiss');

% norm by ctilde
J(n+1,n+1) = -dx*(kfiss')*phi;

% norm by kfiss
J(n+1,7*n+2:8*n+1) = -c_tilde*dx*(phi');

% energy by flux
J(n+2:2*n+1,1:n) = -E*c_tilde*dx;

% energy by ctilde
J(n+2:2*n+1,n+1) = -E*phi*dx;

% energy by energy
J(n+2:2*n+1,n+2:2*n+1) = eye(n);

% energy by kfiss
J(n+2:2*n+1,7*n+2:8*n+1) = -c_tilde*dx*diag(phi);

% temperature by energy
J(2*n+2:3*n+1,n+2:2*n+1) = -R;

% temperature by temperature
J(2*n+2:3*n+1,2*n+2:3*n+1) = S;

% density by temperature
J(3*n+2:4*n+1,2*n+2:3*n+1) = -0.0023393*eye(n);

% density by density
J(3*n+2:4*n+1,3*n+2:4*n+1) = eye(n);

% absorption by density
J(4*n+2:5*n+1,3*n+2:4*n+1) = -DabsxsDrho*eye(n);

% absorption by absorption
J(4*n+2:5*n+1,4*n+2:5*n+1) = eye(n);

% nfiss by density
J(5*n+2:6*n+1,3*n+2:4*n+1) = -DnfissDrho*eye(n);

% nfiss by nfiss
J(5*n+2:6*n+1,5*n+2:6*n+1) = eye(n);

% diff by density
J(6*n+2:7*n+1,3*n+2:4*n+1) = -DdiffDrho*eye(n);

% diff by diff
J(6*n+2:7*n+1,6*n+2:7*n+1) = eye(n);

% kfiss by density
J(7*n+2:8*n+1,3*n+2:4*n+1) = -DkfissDrho*eye(n);

% kfiss by kfiss
J(7*n+2:8*n+1,7*n+2:8*n+1) = eye(n);

% eigenvalue by flux
J(8*n+2,1:n) = -(phi');

% add a 1 to the diagonal
J(8*n+2,8*n+2) = 1;

% form constant preconditioner
```

```
setup.type='nofill';
[L,U] = ilu(J,setup);

end
```

```
function Fun = coupled_steady_res_fun(x)

global geom neut th M F E S R

% construct operators
M = create_operators_steady('M',x);
F = create_operators_steady('F',x);
E = create_operators_steady('E',x);
S = create_operators_steady('S',x);
R = create_operators_steady('R',x);

% get problem size
n = geom.n;
dx = geom.dx;

% get constants from object
%P = th.P; % pressure of system
Qr = th.Qr; % reactor power
Tin = th.Tin; % inlet temperature
absxsREF = neut.absxsREF;
nfissREF = neut.nfissREF;
diffREF = neut.diffREF;
kfissREF = neut.kfissREF;
DabsxsDrho = neut.DabsxsDrho;
DnfissDrho = neut.DnfissDrho;
DdiffDrho = neut.DdiffDrho;
DkfissDrho = neut.DkfissDrho;
rhoREF = th.rhoREF;
Tref = th.Tref;
DrhoDtemp = th.DrhoDtemp;

% allocate function
Fun = zeros(8*n+2,1);

% extract vectors from x
phi = x(1:n);                   % fluxes
c_tilde = x(n+1);               % normalization constant
Q = x(n+2:2*n+1);               % power in cell
T = x(2*n+2:3*n+1);             % cell average temperature
rho = x(3*n+2:4*n+1);           % cell average density
absxs = x(4*n+2:5*n+1);         % absorption macro xs
nfiss = x(5*n+2:6*n+1);         % nu-fission macro xs
diff = x(6*n+2:7*n+1);          % diffusion coefficient
kfiss = x(7*n+2:8*n+1);         % kappa-fission macro xs
lamb = x(8*n+2);                  % eigenvalue
% kfissREF = neut.kfissREF*ones(n,1);

% solve flux equation
Fun(1:n) = M*phi - lamb*F*phi;

% solve Normalization Equation
Fun(n+1) = Qr - c_tilde*dx*kfiss'*phi;

% energy deposition
Fun(n+2:2*n+1) = Q - c_tilde*dx*E*phi;

% temperature calculation
Fun(2*n+2:3*n+1) = S*T - R*Q;
Fun(2*n+2) = Fun(2*n+2) - Tin;
```

```
% density distribution (Table lookup)
%for i = 1:n
%    Fun(3*n+1+i) = rho(i) - XSteam('rho_pT',P,T(i))/1000;
%end
Fun(3*n+2:4*n+1) = (rho - rhoREF) - DrhoDtemp*(T - Tref);

% absorption cross section
Fun(4*n+2:5*n+1) = (absxs - absxsREF) - DabsxsDrho*(rho - rhoREF);

% nu-fission cross section
Fun(5*n+2:6*n+1) = (nfiss - nfissREF) - DnfissDrho*(rho - rhoREF);

% diffusion coefficient
Fun(6*n+2:7*n+1) = (diff - diffREF) - DdiffDrho*(rho - rhoREF);

% kappa-fission cross section
Fun(7*n+2:8*n+1) = (kfiss - kfissREF) - DkfissDrho*(rho - rhoREF);%*kfissconst;

% eigenvalue
Fun(8*n+2) = -1/2*(phi')*phi + 1/2;

end
```

### B.3.1 Analytic Jacobian-vector Product Routine

```
function Jy = JacobianVectMult(x,y)

global geom neut th M F E S R

% create MD operator
MD = create_operators('MD',x);

% get problem size
n = geom.n;
dx = geom.dx;

% allocate function
Jy = zeros(8*n+2,1);

% get perturbation parameter for density
b = 1e-8;
N = length(y(2*n+2:3*n+1));
epsilon = b*sum(x(2*n+2:3*n+1))/(N*norm(y(2*n+2:3*n+1)));

% get constants from object
P = th.P; % pressure of system
DabsxsDrho = neut.DabsxsDrho;
DnfissDrho = neut.DnfissDrho;
DdiffDrho = neut.DdiffDrho;
DkfissDrho = neut.DkfissDrho;

% extract vectors from x
phi = x(1:n);                  % fluxes
c_tilde = x(n+1);              % normalization constant
T = x(2*n+2:3*n+1);            % cell average temperature
rho = x(3*n+2:4*n+1);          % cell average density
kfiss = x(7*n+2:8*n+1);        % kappa-fission macro xs
lamb = x(8*n+2);               % eigenvalue
% kfissREF = neut.kfissREF*ones(n,1);

% phi equation
Jy(1:n) = (M - lamb*F)*y(1:n) + ...
    diag(phi)*y(4*n+2:5*n+1) + ...
    (-lamb*diag(phi))*y(5*n+2:6*n+1) + ...
    MD*y(6*n+2:7*n+1) + ...
    (-F*phi)*y(8*n+2);

% Jy(1:n) = (M - lamb*F)*y(1:n) + (-F*phi)*y(8*n+2);

% flux-power normalization
Jy(n+1) = -c_tilde*dx*(kfiss')*y(1:n) + ...
    (-dx*(kfiss')*phi)*y(n+1)   + ...
    (-c_tilde*dx)*(phi')*y(7*n+2:8*n+1);

% energy deposition
Jy(n+2:2*n+1) = -E*c_tilde*dx*y(1:n)    + ...
    (-E*dx*phi)*y(n+1)     + ...
    eye(n)*y(n+2:2*n+1)  + ...
    (-c_tilde*dx)*diag(phi)*y(7*n+2:8*n+1);

% temperature distribution
Jy(2*n+2:3*n+1) = -R*y(n+2:2*n+1) + ...
    S*y(2*n+2:3*n+1);

% density distribution (approximate with finite difference)
FunXepsY = zeros(n,1);
FunX = zeros(n,1);
for i = 1:n
    FunXepsY(i) = (rho(i)+epsilon*y(3*n+1+i)) - XSteam('rho_pT',P,T(i)+epsilon*y(2*n+1+i))/1000;
    FunX(i) = rho(i) - XSteam('rho_pT',P,T(i))/1000;
end
Jy(3*n+2:4*n+1) = (FunXepsY - FunX)/epsilon + eye(n)*y(3*n+2:4*n+1);

% absorption
Jy(4*n+2:5*n+1) = -DabsxsDrho*eye(n)*y(3*n+2:4*n+1) + ...
    eye(n)*y(4*n+2:5*n+1);

% nu-fission
Jy(5*n+2:6*n+1) = -DnfissDrho*eye(n)*y(3*n+2:4*n+1) + ...
    eye(n)*y(5*n+2:6*n+1);

% diffusion coefficient
Jy(6*n+2:7*n+1) = -DdiffDrho*eye(n)*y(3*n+2:4*n+1) + ...
    eye(n)*y(6*n+2:7*n+1);

% kappa-fission
Jy(7*n+2:8*n+1) = -DkfissDrho*eye(n)*y(3*n+2:4*n+1) + ...
    eye(n)*y(7*n+2:8*n+1);

% eigenvalue
Jy(8*n+2) = -(phi')*y(1:n);

end
```

## B.4 Transient Coupled Neutronics/Thermal Hydraulics

```
function [x_trans,pow,tave,rod] = run_coupled_trans(myfun,mycr,x_steady)

global info geom

% process steady state results into steady object
process_steady(x_steady);

% set steady state unknown vector
x = get_initial_vec();

% get timestep info
Nt = info.time/info.dt;

% get problem size
```

```
n = geom.n;

% create preconditioner
[L,U] = create_precond_trans(x(:,1));

% create vectors
cr = zeros(n,1);
pow = zeros(Nt,1);
tave = zeros(Nt,1);
rod = zeros(Nt,1);

% set initial vals
pow(1) = sum(x(2*n+1:3*n));
tave(1) = sum(x(3*n+1:4*n))/length(x(3*n+1:4*n));
rod(1) = 0;

% begin time loop
for i = 1:Nt

    % change cr
    cr = mycr(i,cr);

    % evaluate function at last time
    myfun_eval = @(xx) myfun(xx,x,cr);
    mymatvecmult_eval = @(x,y) Jacobian_vec_FD(myfun_eval,x,y);

    % iterate to get values at next time step
    x = inexact_newton(myfun_eval,mymatvecmult_eval,x,L,U);

    % calculate power and average temp
    pow(i+1) = sum(x(2*n+1:3*n));
    tave(i+1) = sum(x(3*n+1:4*n))/length(x(3*n+1:4*n));

    % plot
    create_plots(x,pow,tave,i)

    % store rod info
    rod(i+1) = sum(cr);

end

% set x to output
x_trans = x;

end
```

```
function process_steady(x)

global geom steady

% get problem size
n = geom.n;

% get information out of x
steady.phi = x(1:n);
steady.c_tilde = x(n+1);
steady.Q = x(n+2:2*n+1);
steady.T = x(2*n+2:3*n+1);
steady.rho = x(3*n+2:4*n+1);
steady.absxs = x(4*n+2:5*n+1);
steady.nfiss = x(5*n+2:6*n+1);
steady.diff = x(6*n+2:7*n+1);
steady.kfiss = x(7*n+2:8*n+1);
steady.lamb = x(8*n+2);
```

```
end
```

```
function x = get_initial_vec()

global info geom neut steady

% get problem size
n = geom.n;

% get number of time steps
Nt = info.time/info.dt;

% extract from objects
beta = neut.beta;
lambd = neut.lambd;
phi = steady.phi;
lamb = steady.lamb;
nfiss = steady.nfiss;
Q = steady.Q;
T = steady.T;
rho = steady.rho;
absxs = steady.absxs;
diff = steady.diff;
kfiss = steady.kfiss;

% preallocate vector
x = zeros(9*n,1);

% record flux
x(1:n,1) = phi;

% get precursor steady conc and record in vector
x(n+1:2*n,1) = ((beta*lamb)/lambd)*nfiss.*phi;

% set energy deposition
x(2*n+1:3*n,1) = Q;

% set temperatures
x(3*n+1:4*n,1) = T;

% set density
x(4*n+1:5*n,1) = rho;

% set cross sections
x(5*n+1:6*n,1) = absxs;
x(6*n+1:7*n,1) = nfiss;
x(7*n+1:8*n,1) = diff;
x(8*n+1:9*n,1) = kfiss;
```

```
function out_mat = create_operators_trans(oper,x)

global geom neut th

% extract problem size
n = geom.n;

% Find Operator to Create
switch oper

    % Create M operator
    case 'M'
```

```
        % number of nonzeros
        nnz = n + 2*(n-1);

        % preallocate sparse matrix vectors
        rowvec = zeros(nnz,1);
        colvec = zeros(nnz,1);
        valvec = zeros(nnz,1);

        % extract data from data objects
        alb = neut.alb;
        dx = geom.dx;

        % extract needed data from x vector
        D = x(7*n+1:8*n);
        absxs = x(5*n+1:6*n);

        % shift diffusion coefficient vector
        Dm1 = circshift(D,1);
        Dp1 = circshift(D,-1);

        % create diagonal (left boundary, interior, right boundary)
        rowvec(1:n) = 1:n;
        colvec(1:n) = 1:n;

        % left boundary
        valvec(1) = 2/dx*((D(1)*(1-alb(1)))/(4*D(1)*(1+alb(1)) + dx*(1-alb(1))) + ...
            1/dx*(D(1)*D(2))/(D(1) + D(2))) + absxs(1);

        % interior cells
        valvec(2:n-1) = 2/dx^2*((Dm1(2:n-1).*D(2:n-1))./(Dm1(2:n-1) + D(2:n-1)) + ...
            (D(2:n-1).*Dp1(2:n-1))./(D(2:n-1) + Dp1(2:n-1))) + absxs(2:n-1);

        % right boundary
        valvec(n) = 2/dx*(1/dx*(D(n-1)*D(n))/(D(n-1) + D(n)) + (D(n)*(1-alb(2))) / ...
            (4*D(n)*(1+alb(2)) + dx*(1-alb(2)))) + absxs(n);

        % create subdiagonal -1
        rowvec(n+1:2*n-1) = 2:n;
        colvec(n+1:2*n-1) = 1:n-1;
        valvec(n+1:2*n-1) = -2/dx^2*((Dm1(2:n).*D(2:n))./(Dm1(2:n) + D(2:n)));

        % create superdiagonal +1
        rowvec(2*n:3*n-2) = 1:n-1;
        colvec(2*n:3*n-2) = 2:n;
        valvec(2*n:3*n-2) = -2/dx^2*((D(1:n-1).*Dp1(1:n-1))./(D(1:n-1) +Dp1(1:n-1)));

        % create sparse matrix for output
        out_mat = sparse(rowvec,colvec,valvec);

    % Create F operator
    case 'F'

        % number of nonzeros
        nnz = n;

        % extract nfiss from x vector
        nfiss = x(6*n+1:7*n);

        % preallocate sparse matrix vectors
        rowvec = zeros(nnz,1);
        colvec = zeros(nnz,1);
        valvec = zeros(nnz,1);

        % calculate diagonal
        rowvec(1:n) = 1:n;
        colvec(1:n) = 1:n;
        valvec(1:n) = nfiss(1:n);
        % create sparse matrix for output
        out_mat = sparse(rowvec,colvec,valvec);

    % Create E operator
    case 'E'

        % number of nonzeros
        nnz = n;

        % extract kfiss from x vector
        kfiss = x(8*n+1:9*n);

        % preallocate sparse matrix vectors
        rowvec = zeros(nnz,1);
        colvec = zeros(nnz,1);
        valvec = zeros(nnz,1);

        % calculate diagonal
        rowvec(1:n) = 1:n;
        colvec(1:n) = 1:n;
        valvec(1:n) = kfiss(1:n);

        % create sparse matrix for output
        out_mat = sparse(rowvec,colvec,valvec);

    % Create S operator
    case 'S'

        % number of nonzeros
        nnz = n + (n-1);

        % preallocate sparse matrix vectors
        rowvec = zeros(nnz,1);
        colvec = zeros(nnz,1);
        valvec = zeros(nnz,1);

        % calculate diagonal
        rowvec(1:n) = 1:n;
        colvec(1:n) = 1:n;
        valvec(1:n) = 1;

        % calculate subdiagonal -1
        rowvec(n+1:2*n-1) = 2:n;
        colvec(n+1:2*n-1) = 1:n-1;
        valvec(n+1:2*n-1) = -1;

        % create sparse matrix for output
        out_mat = sparse(rowvec,colvec,valvec);

    % Create R operator
    case 'R'

        % number of nonzeros
        nnz = n + (n-1);

        % preallocate sparse matrix vectors
        rowvec = zeros(nnz,1);
        colvec = zeros(nnz,1);
        valvec = zeros(nnz,1);

        % get data from object
        w = th.w;
        cp = th.cp;

        % calculate diagonal
        rowvec(1:n) = 1:n;
        colvec(1:n) = 1:n;
        valvec(1:n) = 1/(2*w*cp);
        % calculate subdiagonal -1
        rowvec(n+1:2*n-1) = 2:n;
        colvec(n+1:2*n-1) = 1:n-1;
        valvec(n+1:2*n-1) = 1/(2*w*cp);

        % create sparse matrix for output
        out_mat = sparse(rowvec,colvec,valvec);

    case 'MD'

        % number of nonzeros
        nnz = n + 2*(n-1);

        % preallocate sparse matrix vectors
        rowvec = zeros(nnz,1);
        colvec = zeros(nnz,1);
        valvec = zeros(nnz,1);

        % extract data from data objects
        alb = neut.alb;
        dx = geom.dx;

        % extract needed data from x vector
        D = x(7*n+1:8*n);
        phi = x(1:n);

        % shift diffusion coefficient and phi vectors
        Dm1 = circshift(D,1);
        Dp1 = circshift(D,-1);
        phim1 = circshift(phi,1);
        phip1 = circshift(phi,-1);

        % create diagonal (left boundary, interior, right boundary)
        rowvec(1:n) = 1:n;
        colvec(1:n) = 1:n;

        % left boundary
        valvec(1) = 2/dx*((dx*(1-alb(1))^2)/(4*D(1)*(1+alb(1)) + dx*(1-alb(1)))^2 + ...
            1/dx*D(2)^2/(D(1) + D(2))^2)*phi(1);

        % interior cells
        valvec(2:n-1) = 2/dx^2*(Dm1(2:n-1).^2./(Dm1(2:n-1) + D(2:n-1)).^2 + ...
            Dp1(2:n-1).^2./(D(2:n-1) + Dp1(2:n-1)).^2).*phi(2:n-1);

        % right boundary
        valvec(n) = 2/dx*((dx*(1-alb(2))^2)/(4*D(n)*(1+alb(2)) + dx*(1-alb(2)))^2 + ...
            1/dx*D(n-1)^2/(D(n-1) + D(n))^2)*phi(n);

        % create subdiagonal -1
        rowvec(n+1:2*n-1) = 2:n;
        colvec(n+1:2*n-1) = 1:n-1;
        valvec(n+1:2*n-1) = -2/dx^2*(Dm1(2:n).^2./(Dm1(2:n) + D(2:n)).^2).*phim1(2:n);

        % create superdiagonal +1
        rowvec(2*n:3*n-2) = 1:n-1;
        colvec(2*n:3*n-2) = 2:n;
        valvec(2*n:3*n-2) = -2/dx^2*(Dp1(1:n-1).^2./(D(1:n-1) + Dp1(1:n-1)).^2).*phip1(1:n-1);

        % create sparse matrix for output
        out_mat = sparse(rowvec,colvec,valvec);

    case default

        error('Operator does not exist!');

end
```

```
end
```

```
function [L,U] = create_precond_steady(x)

global geom neut

% construct operators
M = create_operators_steady('M',x);
F = create_operators_steady('F',x);
E = create_operators_steady('E',x);
S = create_operators_steady('S',x);
R = create_operators_steady('R',x);
MD = create_operators_steady('MD',x);

% get number of meshes
n = geom.n;
dx = geom.dx;

% allocate function%
J = sparse(zeros(8*n+2)); % jacobian of function

% extract vectors from x
DabsxsDrho = neut.DabsxsDrho;
DnfissDrho = neut.DnfissDrho;
DdiffDrho = neut.DdiffDrho;
DkfissDrho = neut.DkfissDrho;

% extract vectors from x
phi = x(1:n);                   % fluxes
c_tilde = x(n+1);               % normalization constant
kfiss = x(7*n+2:8*n+1);         % kappa-fission macro xs
lamb = x(8*n+2);                % eigenvalue

% flux equation by flux
J(1:n,1:n) = (M - lamb*F);

% flux equation by absxs
J(1:n,4*n+2:5*n+1) = diag(phi);

% flux equation by nfiss
J(1:n,5*n+2:6*n+1) = -lamb*diag(phi);

% flux equation by diff
J(1:n,6*n+2:7*n+1) = MD;

% flux by eigenvalue
J(1:n,8*n+2) = -F*phi;

% norm by flux
J(n+1,1:n) = -c_tilde*dx*(kfiss');

% norm by ctilde
J(n+1,n+1) = -dx*(kfiss')*phi;

% norm by kfiss
J(n+1,7*n+2:8*n+1) = -c_tilde*dx*(phi');

% energy by flux
J(n+2:2*n+1,1:n) = -E*c_tilde*dx;

% energy by ctilde
J(n+2:2*n+1,n+1) = -E*phi*dx;

% energy by energy
J(n+2:2*n+1,n+2:2*n+1) = eye(n);
```

```
% energy by kfiss
J(n+2:2*n+1,7*n+2:8*n+1) = -c_tilde*dx*diag(phi);

% temperature by energy
J(2*n+2:3*n+1,n+2:2*n+1) = -R;

% temperature by temperature
J(2*n+2:3*n+1,2*n+2:3*n+1) = S;

% density by temperature
J(3*n+2:4*n+1,2*n+2:3*n+1) = -0.0023393*eye(n);

% density by density
J(3*n+2:4*n+1,3*n+2:4*n+1) = eye(n);

% absorption by density
J(4*n+2:5*n+1,3*n+2:4*n+1) = -DabsxsDrho*eye(n);

% absorption by absorption
J(4*n+2:5*n+1,4*n+2:5*n+1) = eye(n);

% nfiss by density
J(5*n+2:6*n+1,3*n+2:4*n+1) = -DnfissDrho*eye(n);

% nfiss by nfiss
J(5*n+2:6*n+1,5*n+2:6*n+1) = eye(n);

% diff by density
J(6*n+2:7*n+1,3*n+2:4*n+1) = -DdiffDrho*eye(n);

% diff by diff
J(6*n+2:7*n+1,6*n+2:7*n+1) = eye(n);

% kfiss by density
J(7*n+2:8*n+1,3*n+2:4*n+1) = -DkfissDrho*eye(n);

% kfiss by kfiss
J(7*n+2:8*n+1,7*n+2:8*n+1) = eye(n);

% eigenvalue by flux
J(8*n+2,1:n) = -(phi');

% add a 1 to the diagonal
J(8*n+2,8*n+2) = 1;

% form constant preconditioner
setup.type='nofill';
[L,U] = ilu(J,setup);

end
```

```
function Fun = coupled_trans_res_fun(x,x_o,cr)

global info geom neut th steady M F E S R

% construct operators
M = create_operators_trans('M',x);
F = create_operators_trans('F',x);
E = create_operators_trans('E',x);
S = create_operators_trans('S',x);
R = create_operators_trans('R',x);

% get problem size
n = geom.n;
```

```
dx = geom.dx;

% get timestep
dt = info.dt;

% extract previous timestep
phi_o = x_o(1:n);
c_o = x_o(n+1:2*n);
T_o = x_o(3*n+1:4*n);

% get constants from object
%P = th.P; % pressure of system
Tin = th.Tin; % inlet temperature
absxsREF = neut.absxsREF;
nfissREF = neut.nfissREF;
diffREF = neut.diffREF;
kfissREF = neut.kfissREF;
DabsxsDrho = neut.DabsxsDrho;
DnfissDrho = neut.DnfissDrho;
DdiffDrho = neut.DdiffDrho;
DkfissDrho = neut.DkfissDrho;
rhoREF = th.rhoREF;
Tref = th.Tref;
DrhoDtemp = th.DrhoDtemp;
vel = neut.vel;
beta = neut.beta;
lambd = neut.lambd;
lamb = steady.lamb;
c_tilde = steady.c_tilde;
w = th.w;
A = th.area;

% allocate function
Fun = zeros(9*n,1);

% extract vectors from x
phi = x(1:n);
c = x(n+1:2*n);
Q = x(2*n+1:3*n);
T = x(3*n+1:4*n);
rho = x(4*n+1:5*n);
absxs = x(5*n+1:6*n);
nfiss = x(6*n+1:7*n);
diff = x(7*n+1:8*n);
kfiss = x(8*n+1:9*n);

% solve flux equation
Fun(1:n) = phi - phi_o + vel*dt*(M*phi - (1-beta)*lamb*F*phi - lambd*c);

% solve precursor
Fun(n+1:2*n) = c - c_o + dt*(lambd*c - beta*lamb*F*phi);

% energy deposition
Fun(2*n+1:3*n) = Q - c_tilde*E*phi*dx;

% temperature calculation
Fun(3*n+1:4*n) = S*T - R*Q;
Fun(3*n+1) = Fun(3*n+1) - Tin;
Fun(3*n+1:4*n) = T - T_o + (w*dt)./(rho*A*dx).*Fun(3*n+1:4*n);

% density distribution (Table lookup)
% for i = 1:n
%     Fun(4*n+i) = rho(i) - XSteam('rho_pT',P,T(i))/1000;
% end
Fun(4*n+1:5*n) = (rho - rhoREF) - DrhoDtemp*(T - Tref);

% absorption cross section
Fun(5*n+1:6*n) = (absxs - absxsREF) - DabsxsDrho*(rho - rhoREF) - absxsREF.*cr;
```

```
% nu—fission cross section
Fun(6*n+1:7*n) = (nfiss — nfissREF) — DnfissDrho*(rho — rhoREF);

% diffusion coefficient
Fun(7*n+1:8*n) = (diff — diffREF) — DdiffDrho*(rho — rhoREF);

% kappa—fission cross section
Fun(8*n+1:9*n) = (kfiss — kfissREF) — DkfissDrho*(rho — rhoREF);

end
```

### B.4.1 Control Rod Movement File

```
function cr = control_rod(i,cr)

% put CR in
if i >= 50 && i < 100
    if mod(i,5) == 0
        if cr(1) == 0
            cr(1:2) = 1;
        else
            cr(find(cr,1,'last')+1:find(cr,1,'last')+10) = 1;
        end
    end
end

% remove control rod

if i >=500

    if mod(i,5) == 0
            lastidx = find(cr,1,'last')—10;
            if lastidx < 0
                cr(:) = 0;
            else
              cr(find(cr,1,'last'):—1:find(cr,1,'last')—4) = 0;
            end
    end

end


end
```

## C Serpent Input File for Diffusion Theory Parameters

```
################################################################################
#
# Bryan Herman
# 2.29 Numerical Fluid Mechanics
# Term Project
# Pin-cell Calculation -- Density 0.705 g/cc
#
################################################################################


# Define surfaces

surf 1 sqc 0.0 0.0 0.63
surf 2 cyl 0.0 0.0 0.4096
surf 3 cyl 0.0 0.0 0.4178
surf 4 cyl 0.0 0.0 0.4750
```

```
# Define cells

cell fuel 0 mat_fuel -2
cell gap  0 mat_gap  -3 2
cell clad 0 mat_clad -4 3
cell cool 0 mat_cool -1 4
cell out  0 outside   1


# Define materials

mat mat_fuel -10.4
92235.09c   -0.0396676
92234.09c   -0.000317341
92238.09c   -0.841517
8016.09c    -0.118498

mat mat_gap 0.001
2004.06c    1.00

mat mat_clad -6.55
40000.06c  -0.99
41093.06c  -0.01

mat mat_cool -0.705 moder lwtr 1001
1001.06c  2.0
8016.06c  1.0


# Thermal Scattering Library

therm lwtr lwe7.12t


# Cross Section Library

set acelib "/opt/serpent/xs/endfb7/sss_endfb7u.xsdata"


# Set boundary conditions

set bc 3


# Set Cross Section Generation Universe

set gcu 0


# Set Neutron Population

set pop 10000 1000 20


# Set plots

plot 3 500 500
mesh 3 500 500
```