LA-UR-11-04834

*Approved for public release;*
*distribution is unlimited.*

| | |
|---|---|
| *Title:* | Coarse Mesh Finite Difference in MCNP5 |
| *Author(s):* | Mitchell T.H. Young, Forrest B. Brown, Brian C. Kiedrowski, William R. Martin |
| *Intended for:* | MCNP References |




Form 836 (7/06)

# Coarse Mesh Finite Difference in MCNP5

Mitchell T.H. Young F.B. Brown* B.C. Kiedrowski*
W.R. Martin

The University of Michigan–Ann Arbor
Department of Nuclear Engineering & Radiological Sciences
2355 Bonisteel Boulevard, Ann Arbor, MI 48109, USA
youngmit@umich.edu, wrm@umich.edu

*Los Alamos National Laboratory
X-Computational Physics Division, Monte Carlo Codes Group
P.O. Box 1663, MS A143
Los Alamos, NM 87545, USA
fbrown@lanl.gov, bckiedro@lanl.gov

August 18, 2011

**Abstract**

Lee, et. al.[4] [5] have demonstrated the feasibility of applying a Coarse Mesh Finite Difference (CMFD) acceleration technique to accelerate fission source distribution (FSD) convergence in monte carlo criticality calculations. Most of these implementations have been done in 1- and 2-D with multigroup monte carlo. In this work, a CMFD solver has been implemented in MCNP to facilitate FSD acceleration in 3-D with continuous-energy cross sections for more general applications. Some promising results have been obtained for full-core reactor simulations in which pure finite difference techniques have been able to accelerate FSD convergence. CMFD results have proved less robust and require further investigation.

# Contents

# 1 Introduction

A new module, `cmfd_mod` is added to the MCNP5 source code to provide coarse mesh finite difference (CMFD) functionality for $k$-eigenvalue calculations. CMFD can be used to accelerate the convergence of the fission source distribution in critical systems with high dominance ratios, such as certain large reactors or larger systems like spent fuel pools.

The CMFD solver uses the data from several mesh tallies to generate 2-group cross sections of interest (see section 2) and uses these cross sections to solve a set of linear equations to converge the fission source deterministically on a coarse mesh. If the fission source distribution is significantly more accurate than the current state of the Monte Carlo-determined fission source, acceleration of the fission source convergence can be achieved.

# 2 CMFD Theory

The CMFD implementation in MCNP operates by inserting an extra step in between the KCODE cycles that performs the following tasks:

1. Compute multigroup cross sections within each mesh region,
2. initialize the CMFD equations,
3. solve the system iteratively, and
4. resample the fission source bank using CMFD results.

How each of these tasks are performed will be described from a theoretical standpoint below.

## 2.1 Multigroup Cross Sections

While in most cases, MCNP operates on continuous-energy cross-section data, the CMFD equations are multigroup and therefore require multigroup cross sections. More specifically, while the solution routines have been generalized for an arbitrary number of energy groups, a two-group formulation is presently employed. Two-group cross sections are much simpler to implement, since the scattering matrix only contains an entry for $\Sigma_{s12}$, which in the absence of upscattering can be calculated using a balance equation with already available data. Obtaining a full multigroup scattering matrix would require additional modifications to the MCNP code to tally each inter-group scattering event. A simple balance equation for group 2 neutrons is instead used to solve for $\Sigma_{s12}$ directly,

$$\Sigma_{s12}\phi_1 = \Sigma_{a2}\phi_2 + J^+_{net,2} - J^-_{net,2}, \tag{1}$$

where $\Sigma_{a2}$ is the absorption cross section in the thermal group; $\phi_1$ and $\phi_2$ are the fast and thermal fluxes, respectively; $J^+_{net,2}$ is the net outgoing thermal current from a control volume; and $J^-_{net,2}$ is the net incoming thermal current.

The other cross sections of interest in the CMFD equations are the removal cross section, $\Sigma_r$, the fission production cross section $\nu\Sigma_f$, and the diffusion coefficient, $D$. In this analysis the diffusion coefficient is approximated by assuming isotropic scatter, giving the definition

$$D = \frac{1}{3\Sigma_t} \tag{2}$$

In the two-group case it is assumed that all fission neutrons are born into group 1, and therefore a $\chi$ distribution is not needed.

Cross section values are determined within each mesh region of the CMFD problem domain using FMESH tallies and corresponding tally multipliers for the reactions of interest (total, $\nu\Sigma_f$, absorption). The associated tally scores are accessed directly from the corresponding `fm` array internal to MCNP. Since these tally

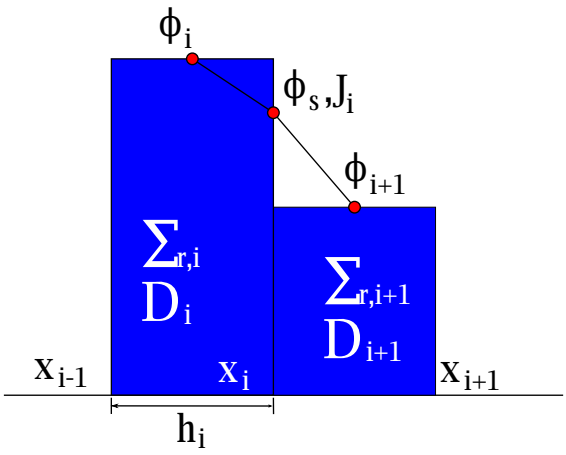



Figure 1: Box scheme in 1 [3]

scores have not been normalized by volume or source particle weight at the time of cross section calculation, the quantity contained in the score is

$$\text{tally} = \int_V \phi \Sigma dV, \tag{3}$$

where $\Sigma$ is a cross section of interest and $\phi$ is the scalar flux within region $V$.

It is possible to calculate a cross section of interest as a ratio of the un-normalized tally scores for an FM multiplied reaction rate tally and an un-multiplied flux tally,

$$\Sigma = \frac{\text{reaction tally}}{\text{flux tally}}. \tag{4}$$

The removal cross section in the context of the CMFD equations is defined as $\Sigma_t - \Sigma_{sgg}$. Since MCNP does not collect scattering matrix data, an alternate formulation is used which considers absorption and out-scattering as

$$\Sigma_{r1} = \Sigma_{a1} + \Sigma_{s12} \tag{5}$$

$$\Sigma_{r2} = \Sigma_{a2}. \tag{6}$$

## 2.2 CMFD Formulation

CMFD is a method derived from standard finite difference diffusion theory, which uses neutron currents obtained from a higher-order solution to improve the fidelity of the finite difference (FDM) solution. In standard FDM, Fick's Law is used to determine the current between two elements in the problem domain. Figure 1 depicts a one-dimensional representation of the box scheme used below.

Current at the interface between the $i$th and $i+1$th can be expressed using Fick's Law from the left and right sides as

$$J_{s,l} = \frac{-D_i(\phi_s - \phi_i)}{h_i/2}, \text{ and} \tag{7}$$

$$J_{s,r} = \frac{-D_{i+1}(\phi_{i+1} - \phi_s)}{h_{i+1}/2}. \tag{8}$$

By imposing equality between the surface currents as determined from the right and left sides, the expression

$$\frac{-D_i(\phi_s - \phi_i)}{h_i/2} = \frac{-D_{i+1}(\phi_{i+1} - \phi_s)}{h_{i+1}/2} \tag{9}$$

is obtained. Solving for $\phi_s$ [3] yields

$$\phi_s = \frac{\frac{D_i}{h_i}\phi_i + \frac{D_{i+1}}{h_{i+1}}\phi_{i+1}}{\frac{D_i}{h_i} + \frac{D_{i+1}}{h_{i+1}}}. \tag{10}$$

Defining a new term, *relative diffusivity* $\beta$ as

$$\beta_i = \frac{D_i}{h_i}, \tag{11}$$

and rearranging the expression for $J_i$ gives the result

$$J_i = -\frac{2\beta_i\beta_{i+1}}{\beta_i + \beta_{i+1}}(\phi_{i+1} - \phi_i). \tag{12}$$

Finally a coupling coefficient, $\tilde{D}_i$ is defined as

$$\tilde{D}_i = \frac{2\beta_i\beta_{i+1}}{\beta_i + \beta_{i+1}}, \tag{13}$$

which is a quantity that relates the surface current, $J_i$ to the flux difference between the $i$- and $i-1$th mesh regions. With the above definitions in place, it is straightforward to develop a neutron balance equation in one dimension is developed by considering loss and source terms. In the multigroup case, the balance equation for a mesh region with neighbors to the left and right are expressed as

$$h_i\Sigma^i_{r,g}\phi^i_g - \tilde{D}^i_g(\phi^{i+1}_g - \phi^i_g) = F^i_g + S^i_g - \tilde{D}^i_g(\phi^i_g - \phi^{i-1}_g), \tag{14}$$

where $S^i_g$ and $F^i_g$ are the total scattering and fission sources for the $i$th region within group $g$, respectively:

$$S^i_g = V_i \sum_{g' \neq g} \Sigma^i_{sg'g}\phi^i_{g'} \tag{15}$$

$$F^i_g = V_i \frac{\chi^i_g}{k} \sum_{g' \in G} \nu\Sigma^i_{fg'}\phi^i_{g'}. \tag{16}$$

In the one-dimensional case, the mesh volumes, $V_i$ are treated as mesh widths, $h_i$.

### 2.2.1 CMFD Correction

The concept of CMFD is introduced into the standard FDM equations. CMFD operates by introducing an extra term, $\hat{D}$ to the current equation to produce

$$J_i = -\tilde{D}_i(\phi_{i+1} - \phi_i) + \hat{D}_i(\phi_i + \phi_{i+1}). \tag{17}$$

The value of $\hat{D}$ is obtained from a higher-order solution. Rewriting Eq. (14) with the inclusion of the $\hat{D}$ correction, and performing some rearrangement yields

$$\phi^{i-1}_g(-\tilde{D}^{i-1}_g - \hat{D}^{i-1}_g) + \phi^i_g(h_i\Sigma^i_{r,g} + \tilde{D}^{i-1}_g + \tilde{D}^i_g + \hat{D}^i_g - \hat{D}^{i-1}_g) + \phi^{i+1}_g(-\tilde{D}^i_g + \hat{D}^i_g) = S^i_g + F^i_g. \tag{18}$$

The above equation is arranged so that all terms corresponding to each flux value are collected. This formulation is more similar to the matrix representation of the system, which is presented elsewhere in this document.

### 2.2.2 Extension to 3D

For use in real-world applications, it is necessary to extend equation (18) to three dimensions. Fortunately, this is quite simple, since all that is needed is the addition of extra coupling coefficients to account from mesh interfaces in the $y$ and $z$ directions. From here on, different notation is used to reference neighboring nodes in each direction; each neighbor will be referenced as north/south ($y$ direction), east/west ($x$ direction), and up/down ($z$ direction). For neutron current conventions, the positive direction is considered to be west→east, north→south, and top→bottom. The subscripts $n$, $s$, $e$, $w$, $u$,and $d$ are used to denote these directions. Incorporating all three dimensions results in the balance equation

$$\begin{aligned}
&\phi_g^w A_x(-\tilde{D}_g^w - \hat{D}_g^w) + \phi_g^e A_x(-\tilde{D}_g^e + \hat{D}_g^e) + \phi_g^i A_x(\tilde{D}_g^w + \tilde{D}_g^e + \hat{D}_g^e - \hat{D}_g^w) + \\
&\phi_g^n A_y(-\tilde{D}_g^n - \hat{D}_g^n) + \phi_g^s A_y(-\tilde{D}_g^s + \hat{D}_g^s) + \phi_g^i A_y(\tilde{D}_g^n + \tilde{D}_g^s + \hat{D}_g^s - \hat{D}_g^n) + \\
&\phi_g^u A_z(-\tilde{D}_g^u - \hat{D}_g^u) + \phi_g^d A_z(-\tilde{D}_g^d + \hat{D}_g^d) + \phi_g^i A_z(\tilde{D}_g^u + \tilde{D}_g^d + \hat{D}_g^d - \hat{D}_g^u) + \\
&V_i \phi_g^i \Sigma_{r,g}^i = S_g^i + F_g^i,
\end{aligned} \tag{19}$$

where $A_x$, $A_y$ and $A_z$ are the cross sectional areas of the mesh elements perpendicular to the $x$, $y$ and $z$-axes, respectively.

Equation (19) is solved iteratively in a large coupled system of equations by the `cmfd_solve` routine.

### 2.2.3 Boundary Conditions

The boundary of the spatial domain is handled using an albedo boundary condition. In this formulation the albedo, $\alpha$ is defined as

$$\alpha = \frac{-J_s}{\phi_s}, \tag{20}$$

where $J_s$ is the incoming current at a boundary surface and $\phi_s$ is the flux on the boundary surface. By employing Fick's Law to represent the $J_s$ in terms of the the flux in the mesh region containing the boundary surface, the following expression is obtained:

$$J_s = -\alpha_s \phi_s = -D_i \frac{\phi_i - \phi_s}{\frac{h_i}{2}}, \tag{21}$$

where $D_i$, $\phi_i$ and $h_i$ are the diffusion coefficient, flux and width of the boundary mesh region, respectively. Solving for $\phi_s$ yields

$$\phi_s = \phi_i \left( \frac{\frac{2D_i}{h_i}}{\frac{2D_i}{h_1} + \alpha_s} \right). \tag{22}$$

Dividing the numerator and denominator by two allows the use of our definition $\beta_i \equiv \frac{D_i}{h_i}$ to assume a more familiar form. Defining a boundary diffusivity, $\beta_s = \frac{\alpha_s}{2}$ results in

$$\phi_s = \phi_i \left( \frac{\frac{D_i}{h_i}}{\frac{D_i}{h_i} + \frac{\alpha_s}{2}} \right) = \phi_i \left( \frac{\beta_i}{\beta_i + \beta_s} \right). \tag{23}$$

Inserting the above expression for $\phi_s$ into Eq. (21) produces

$$J_s = -\alpha_s \frac{\beta_i}{\beta_i + \beta_s} \phi_i = \frac{2\beta_s \beta_i}{\beta_s + \beta_i}. \tag{24}$$

The above result looks suspiciously like our previous definition of $\tilde{D}$, allowing us to treat boundary surfaces similarly to interior surfaces by calculating $\tilde{D}$ for the boundary using the modified definition for the surface diffusivity, $\beta_s$. Once the coupling coefficients, $\tilde{D}_s$ have been generated, the only difference in treatment from interior surfaces is that only the flux of the boundary mesh region is used to calculate the current;

$$J_s = \tilde{D}_s \phi_i. \tag{25}$$

# 3 Functionality

## 3.1 Module Installation

Building MCNP5 with CMFD support is relatively simple. Following the steps below and rebuilding MCNP will result in a new MCNP build with the CMFD functionality.

1. Copy the `cmfd_mod.F90` module file to the `src/` directory.
2. Copy the updated `fmesh_mod.F90` module file into the `src/` directory.
3. Replace the `Depends` file with the included version. This adds the `cmfd_mod` module as a dependency to several other source files.
4. Edit `crit1_mod.F90` to include
   (a) a `USE` statement for the CMFD module (`USE ::  cmfd_mod, ONLY cmfd_test`) in the preamble,
   (b) a call to the `cmfd_test` subroutine in the location shown in Listing 1.

Listing 1: Calling cmfd_test

```
293     ! reorder fso by history number if threading requires it.
294     if( ntasks>1 ) then
295       ! fission bank data in fso_src,  use fso_bnk for scratch
296       call fso_reorder( fso_max_items,fso_max_count, fso_src_count, fso_src,   fso_bnk )
297     endif
298
299 call cmfd_test(1.0)
300
301     ! turn off flag if settling cycles are all done.
302     if( kcy==ikz .and. kcheck==0 )  cpk = cts
303     call ra_iichck(mcheck)
304     if( kcheck>0 .and. kcy-1==lsav .and. mcheck==0 )  cpk = cts
305     if( ksdef/=0 )  ksdef = -1
306   endif
```

## 3.2 Module Initialization

Every time the CMFD solution routine is called, it checks the value of a logical variable that indicates the initialization status of the module. If the module is uninitialized, the subroutine `cmfd_init` is called. The purpose of the subroutine is to collect basic information about the MCNP problem that is being run and to allocate memory for all of the internal variably-sized arrays.

The `cmfd_init` subroutine first sweeps through the `fm` array (the primary storage location for FMESH tallies), searching for internal IDs of the FMESH tallies that are needed for cross section generation and partial currents. Once all of the FMESH tallies have been located, the geometric characteristics of the meshes are inspected to determine the geometry of the CMFD problem domain. The number of mesh elements in each direction as well as the size of each mesh region along each axis is determined from the mesh used to determine partial currents. MCNP assumes that all other meshes are geometrically identical, and there is currently no error checking functionality to verify this.

## 3.3 FMESH Tallies

The correct implementation of FMESH tallies in the MCNP input deck is essential to the proper functionality of the CMFD module. FMESH tallies are needed for

- partial currents,

Table 1: Magic numbers and FM cards for each FMESH tally. Interaction numbers assume continuous energy.

| Interaction | FMESH Number | FM Card[6] |
|---|---|---|
| Partial current | 1 [a] | None |
| Flux | 4 | None |
| Total | 14 | FM -1.0 0 -1 |
| Absorption | 34 | FM -1.0 0 -2:-6 |
| Fission | 24 | FM -1.0 0 -6 -7 |

[a] Any number ending in a 1 will result in a partial current tally.

- un-multiplied neutron flux,
- total interactions,
- absorption interactions, and
- fission-neutron generation ($\nu\Sigma_f$).

Since the partial current FMESH tally only tallies outward currents for each mesh element, it is necessary to specify the mesh to include a "halo" of inactive elements surrounding the active problem domain in order to properly capture incoming current at the boundary of the problem domain. The CMFD module automatically discards the data in these ghost mesh elements. In order to maintain consistency between the current mesh and the other meshes, this halo should be incorporated into the the other meshes as well. In the end, all FMESH cards should have the *exact same* geometric specifications.

While this functionality may be changed in the future, the CMFD module's initialization subroutine currently uses several "magic numbers" to locate the necessary mesh tallies. The numbers used for each mesh tally are presented in Table 1.

### 3.4 Updating

Following initialization, the CMFD module has no values for the multigroup cross sections. It is necessary to run the `cmfd_update` subroutine to obtain values from the mesh tallies and calculate necessary cross sections. The update routine sweeps through each active mesh element interface and stores the value in the associated current mesh tally and normalizes it by the total source particle weight and the area of the interface current. The resultant current value is calculated with

$$J = FM/(AW_{tot}), \tag{26}$$

where $FM$ is the tally score, $A$ is the interface area and $W_{tot}$ is the total source particle weight.

Another sweep is then performed to calculate cross sections within each region. The raw tally score from each mesh tally is divided by the raw score of the un-multiplied flux tally to produce the cross section for the interaction of interest. Once the tallied cross sections are calculated, the downscattering cross section, $\Sigma_{s12}$ is generated using

$$\Sigma_{s12} = \frac{A_2 + J_2^{out} - J_2^{in}}{\Phi_1}, \tag{27}$$

where $A_2$ is the un-normalized absorption-multiplied tally score for the thermal group and $\Phi_1$ is the un-normalized fast group flux. This formula is adapted from Eq. (1).

The removal cross sections for the fast and thermal group are calculated as described in Eqs. (5) and (6).

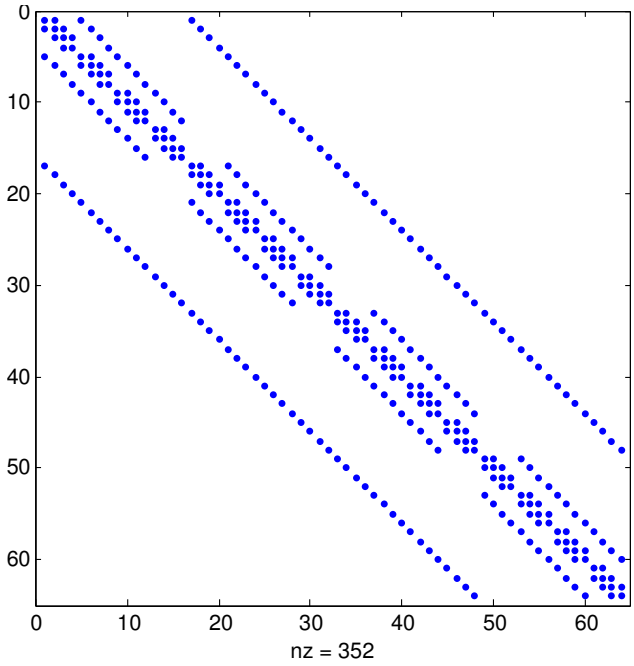



Figure 2: Structure of the migration matrix, **M**. Bold lines indicate non-zero entries, while horizontal and vertical lines are included to show structure.

With all group constants accounted for, the update routine proceeds to calculate the relative diffusivities, $\beta$. Surfaces at the boundary of the mesh are treated differently, using a surface albedo (see section 2.2.3),

$$\beta_{bound} = \frac{J^{in}_{bound}}{2\phi}. \tag{28}$$

The $\tilde{D}$ and $\hat{D}$ values associated with each mesh interface are then calculated using these $\beta$ values. The expression for $\hat{D}$ is a simple rearrangement of Eq. (17), giving

$$\hat{D} = \frac{J + \tilde{D}(\phi_R - \phi_L)}{\phi_R + \phi_L}, \tag{29}$$

where $\phi_R$ and $\phi_L$ correspond to the flux in the mesh elements to the right and left (with respect to positive current conventions) of the surface for which $\hat{D}$ is being calculated.

Finally, the update subroutine constructs a migration matrix (discussed more in section 3.5) by calculating each of the terms in Eq. (19) multiplied by each flux. Terms corresponding to neighboring fluxes are stored in the `coup` array, as they comprise the off-diagonals of the migration matrix. The diagonal of the migration matrix, which contains all terms multiplied by the local mesh region flux, is stored in a separate vector called `diag`.

## 3.5 CMFD Calculation

Solution of the CMFD equations is carried out by applying the power method to the matrix equation

$$\mathbf{M}\phi = (\lambda\mathbf{F} + \mathbf{S})\,\phi, \tag{30}$$

where **M** is the migration matrix defined by the left-hand side of Eq. (19), **F** and **S** are the fission and scattering matrices represented on the right-hand side of Eq. (19). As it is implemented, the solution mechanism of the CMFD module employs a group-major ordering scheme, in which the flux vector and matrices

use energy group as the primary ordering index, and node index as the secondary index. Furthermore, the nodes are indexed using the "natural" ordering scheme, in which the node index, $i$ is determined using

$$i = x_{max} y_{max} z_i + x_{max} y_i + x_i, \tag{31}$$

where $x_{max}$ and $y_{max}$ are the number of nodes along the $x$- and $y$-axes, respectively, and $x_i$, $y_i$, and $z_i$ are the coordinates of node $i$. With these definitions, the migration matrix has the form of the banded septi-diagonal matrix shown in Fig 2. The main diagonal contains the terms multiplied by the current mesh region flux, and the off-diagonals contain the coupling coefficients between the current mesh region and its neighbors. Gaps in the migration matrix occur at nodes which lie on the boundary of the mesh, since they are not coupled to any neighboring node. Leakage through the boundary of the problem domain is accounted for in the diagonal term of the matrix. In its natural form, this system would be very computationally challenging to solve. To help simplify the problem, the far off-diagonals (north, south, up, and down coupling) are subtracted to the right hand side of the equation and incorporated into the solution routine using a previous iteration flux value. The remaining block tri-diagonal matrix is then solved directly using LU decomposition with forward-backward substitution to compute the flux in the current group along a single strip of mesh regions along the $x$-axis.

The sweep along the $y$- and $z$-axes which performs this operation on the entire domain is referred to as the *inner iteration*, which is repeated several times to achieve partial convergence for the current fission and scattering sources that are fixed for the duration of the inner iterations. Following the series of inner iterations, the next energy group is selected and the inner iterations are performed for that group. Once all groups have been operated upon, the fission and scattering sources are updated and the process is repeated. This level of repetition is called the *outer iteration*, and is repeated until the $k$ and the flux distribution have converged, or a maximum number of outer iterations have been performed. The entire soution routine is depicted in Fig. 3.

### 3.6 Fission Source Redistribution

Following the convergence of the flux distribution from the power method, the solution is used to redistribute the fission source bank that MCNP uses to sample source neutrons in the following cycle. A fission source distribution is first calculated using the multi-group flux distribution using

$$\psi^i = \sum_{g \in G} \nu \Sigma_{fg}^i \phi_g^i, \tag{32}$$

where $\psi^i$ is the fission source within mesh region $i$. Once the fission source is calculated, the relative strengths are used as sampling weights to bias the selection of the fission neutrons already contained within the `fso_bnk` array. To perform this weighting the `fso_bnk` array is swept to determine the source point population within each mesh element. A weighting vector (one entry per entry in `fso_bnk`) is then generated by taking the ratio of the fission source strength to the source point population of the mesh region,

$$W^i = \frac{\psi^i}{N^i}, \tag{33}$$

where $N^i$ is the number of fission source points located in the $i$-th node of the monte carlo-generated fission source bank and $\psi^i$ is the fission source strength determined from the CMFD calculation.

Following the generation of the weight vector, the subroutine `cmfd_sample` is used to perform weighted sampling of the existing `fso_bnk` array. A single-pass method is used to draw source points from `fso_bnk` and store them in `fso_src` based on their weights. The sampling routine uses a random process to sample each point in `fso_bnk` a number of times consistent with that point's weight. The output of the routine is a list of indexes corresponding to entries in `fso_bnk`. This list is used to construct `fso_src` with the new fission source distribution.

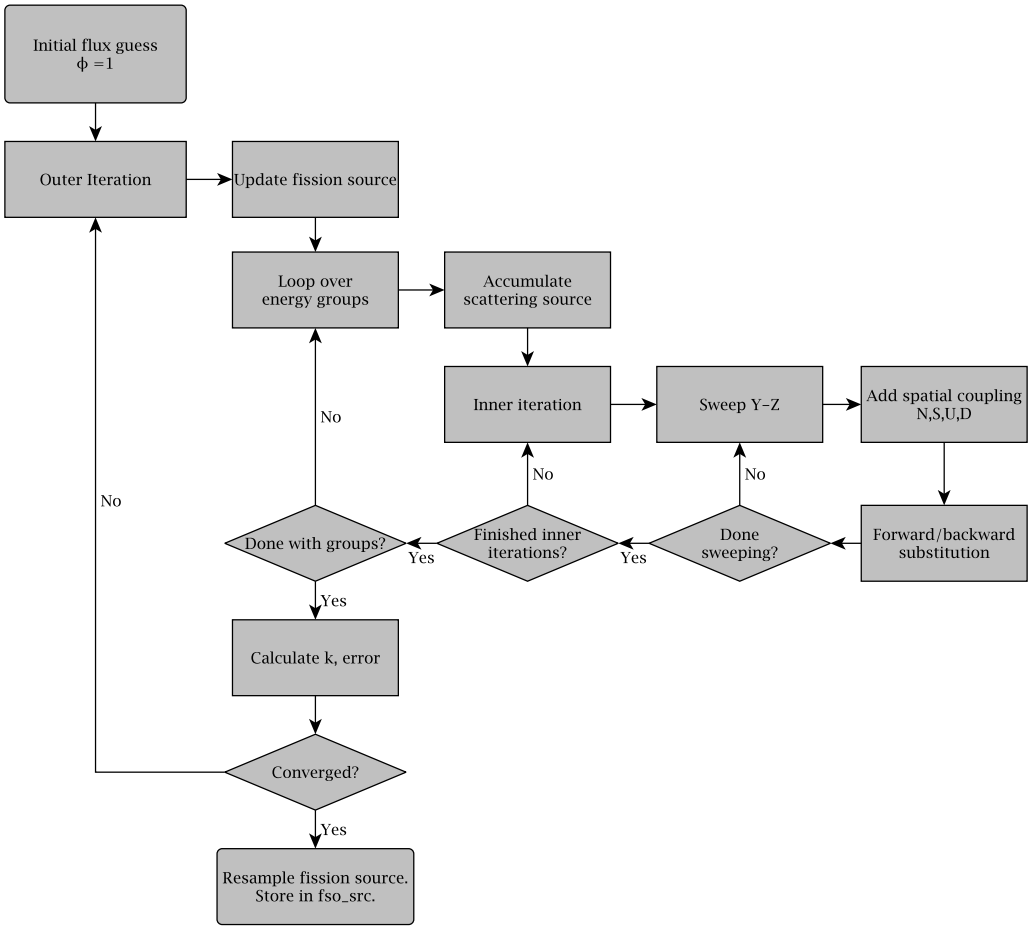



Figure 3: Simplified solution routine.

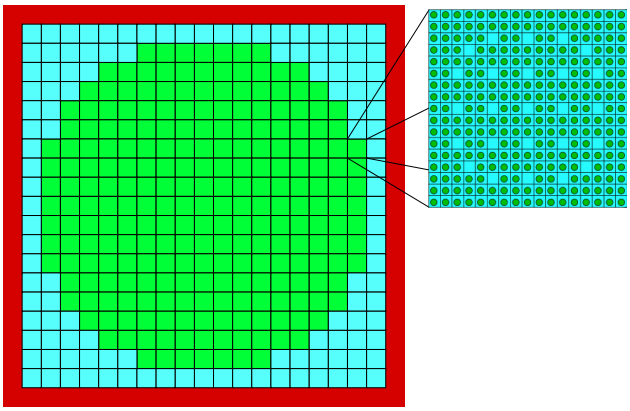


Figure 4: Top view of the boxy Kord Smith Challenge problem.

# 4 Results

## 4.1 Boxy Kord Smith Challenge

An adaptation of the "Kord Smith Challenge" problem [2] was created to test the functionality and effectiveness of the CMFD method. The original benchmark describes a full-core system with a cylindrical reactor vessel, which results in several regions of the solution mesh to lie in void regions, which present difficulties to the CMFD solver. To avoid issues related to these void mesh elements, the problem was modified to have a square-shaped reactor vessel with water filling the extra space. A top view of the modified problem is depicted in Fig. 4.

### 4.1.1 Pure FDM Results

It was found that the $\hat{D}$ term used for CMFD correction is highly sensitive to stochastic noise. To achieve a stable solution with fewer histories, pure FDM (no CMFD correction) was used to obtain the following results. The calculation used 400,000 particles per cycle, and 10 cycles before performing the FDM calculation.

Values of the Shannon entropy of the fission source distribution at each cycle are plotted in Fig. 5 for both the FDM-accelerated case and the natural case, in which no acceleration technique is used. At cycle 10, the adjustment of the fission source results in a distribution that it much closer to the converged FSD. There is an undershot of the converged source entropy from which the fission source must recover, however source convergence is accelerated significantly nonetheless. The FDM case appears to have converged after about 50 cycles, while the natural case requires upwards of 100 cycles to converge.

Figure 6 depicts the thermal flux distribution on a plane normal to the $z$-axis in the middle of the core before and after FDM correction. By the 10th cycle, the flux distribution is still very flat from the initial guess and has yet to develop the center-peaked distribution that is anticipated. Immediately after the FDM correction, the flux distribution assumes this shape.

### 4.1.2 CMFD Results

Analyzing the same problem using the same KCODE settings (400,000 particles per cycle and 10 cycles before CMFD) with the CMFD correction enabled appeared highly sensitive to stochastic noise in the tally data. Figure 7 presents the CMFD solution for thermal flux from six independent runs using different random number seeds. Clearly the solution is highly variant, which implies that the values for $\hat{D}$ are not fully converged. Direct comparison of the $\hat{D}$ values from 10 independent runs showed an average relative

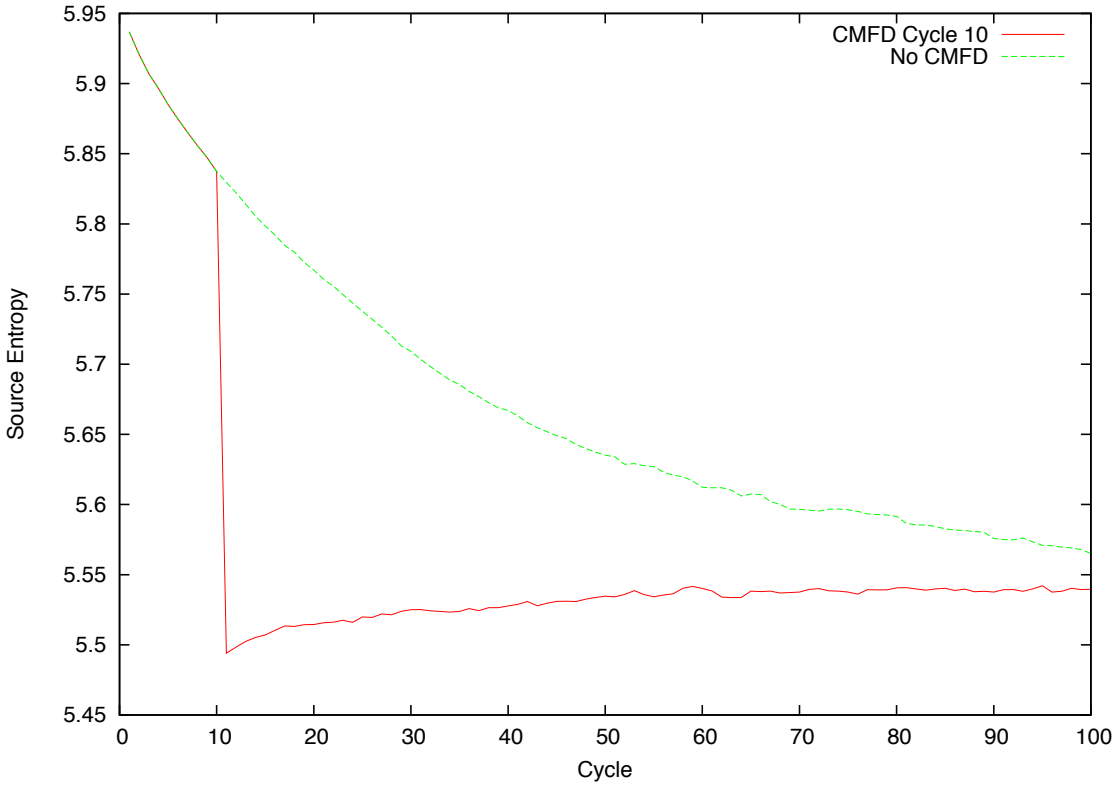
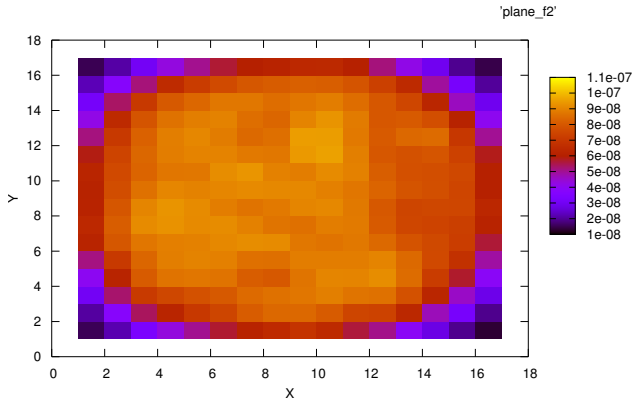
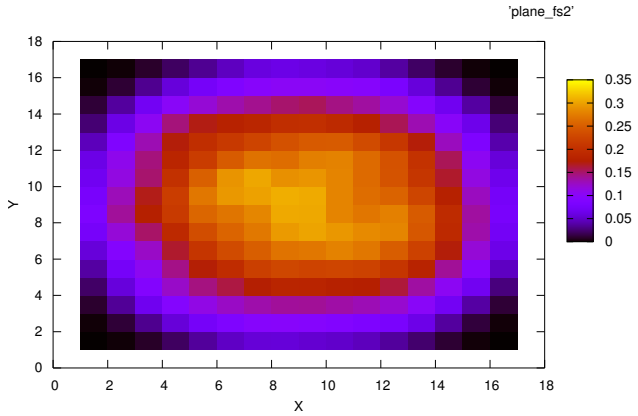



Figure 5: Shannon entropy convergence of the fission source distribution with the FDM solver invoked between the 10th and 11th cycles.




(a) Before FDM




(b) After FDM

Figure 6: Thermal flux distribution before and after FDM correction for a plane halfway up the $z$-axis.

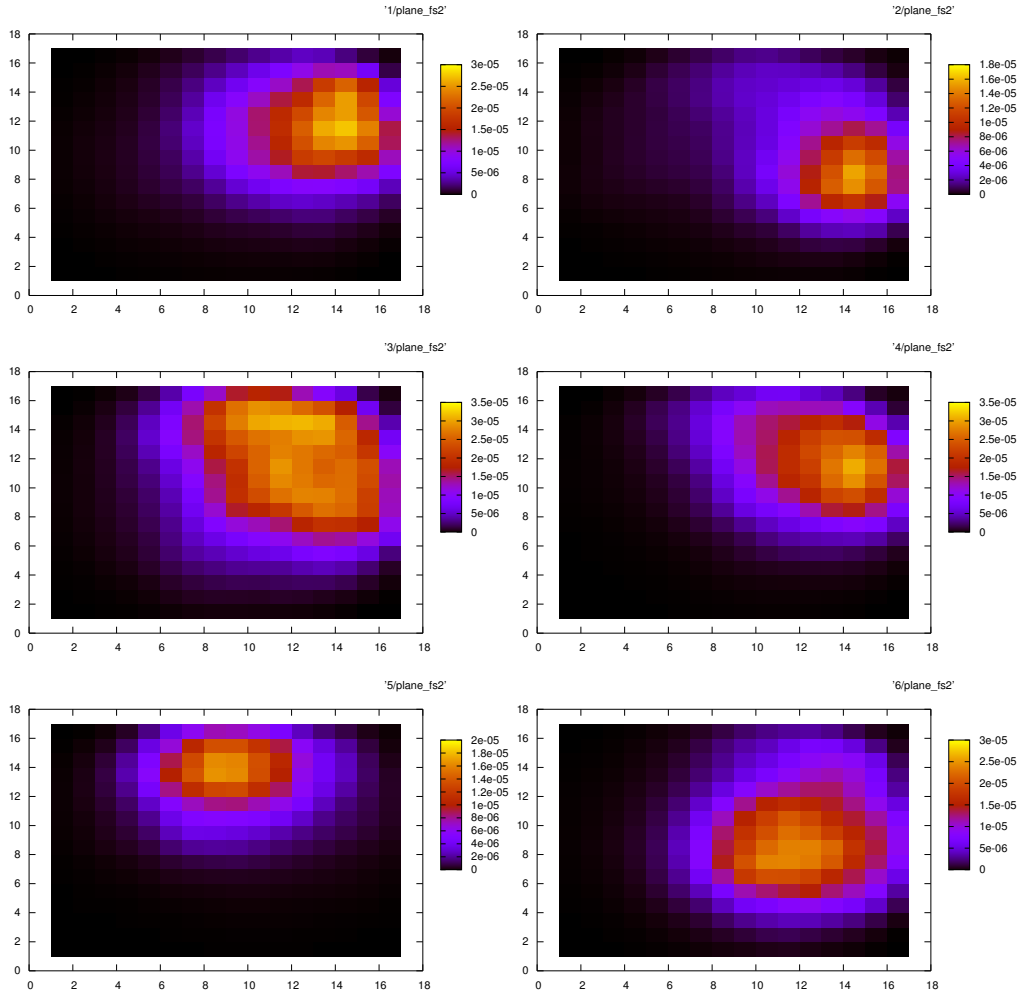
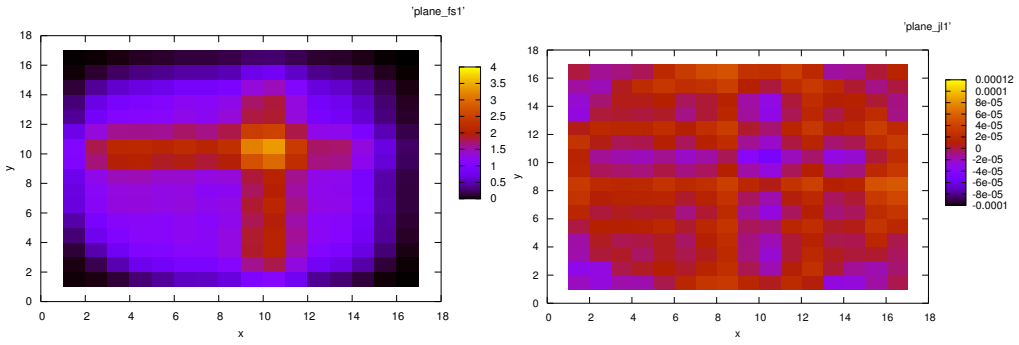



Figure 7: CMFD solution flux distributions from several independent runs.

standard deviation of 2.0479. From these results, $\hat{D}$ appears to be much more variant than the most other group constants, which have average relative standard deviations in the range of several percent.

Using many more particles per cycle and accumulating 30 cycles worth of tallies resulted in a much better result, though still obviously incorrect. Figure 8 depicts the fast flux and net leakage from the CMFD solution using 800,000 particles per cycle and 30 cycles of tally data before calculation. While the result is much closer to the right solution than the previous case, there still appear to be inconsistencies. The cross-shaped flux peaking is non-physical and appears to be an artifact of the current tally. Figure 8b shows the net leakage from each mesh element, with positive values representing a loss of neutrons to the element's surroundings and negative values representing a source of neutrons from the element's surroundings. Since the CMFD correction aims to preserve this current scenario, it is not surprising to see flux peaking in elements with abnormally high incoming current.




(a) Fast flux (b) Fast leakage

Figure 8: CMFD solution for fast flux and net leakage.

# 5 Conclusions

Results obtained with the pure FDM methods appear promising for the case that was examined. The FSD produced by the solver was significantly closer to being fully converged than the original distribution, aiding in the rate of convergence. In most cases however, the fission source sampling routine tended to chronically over-concentrate the FSD when building the fission source bank for the next cycle. This behavior results in a sampled FSD that has a notably lower Shannon entropy than the actual solution obtained from the FDM/CMFD solver. This behavior is undesirable, since it impairs the efficiency of the method and prevents the effectiveness of using the method multiple times throughout the inactive cycles.

Implementation of the CMFD correction was found to be very sensitive to stochastic noise, exhibiting much higher variance than other group constants used to solve the FDM system.

Future work should involve further investigation of the sensitivity of the CMFD correction, and analysis of the potential payoff of using the number of particles needed to accurately employ the method. A more broad survey of the types of problems in which the CMFD method is applicable or beneficial would also be helpful. Other methods of sampling the FSD for use as a monte carlo source should also be explored in the hopes of more accurately portraying the deterministic solution with the fission source bank.

In the case of 3D, continuous-energy monte carlo it is possible that CMFD is simply not an effective method. Instead, investigation of the FDM without CMFD correction could lead to a more promising technique. Without the CMFD correction it would likely be necessary to employ a similarly coarse mesh for obtaining group constants, but a finer mesh for calculating a solution in order to minimize discretization error.

# References

[1] Bruce A. Finlayson. LU decomposition of a tridiagonal matrix. *http://faculty.washington.edu/finlayso/ebook/algebraic/advanced/LUtri.htm*.

[2] J.E. Hogenboom, W.R. Martin, and B. Petrovic. Monte carlo performance benchmark for detailed power density calculation in a full size reactor core. *OECD/NEA*, 2009.

[3] H. G. Joo. Solution of one-dimensional, one-group neutron diffusion equation. *NERS 561 Lecture Notes. University of Michigan.*, 2011.

[4] Min-Jae Lee, Han Gyu Joo, Deokjung Lee, and Kord Smith. Investigation of cmfd accelerated monte carlo eigenvalue calculationi with simplified low dimensional multigroup formulation. *PHYSOR*, 2010.

[5] Min-Jae Lee, Han Gyu Joo, Deokjung Lee, and Kord Smith. Multigroup monte carlo reactor calculation with coarse mesh finite difference formulation for real variance reduction. *Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo*, 2010.

[6] X-5 Monte Carlo Team. *MCNP A General Monte Carlo N-Particle Transport Code, Version 5 Volume II: Users Guide*. Los Alamos National Laboratory, LA-CP-03-0245, February 2008.

# A Subroutine and Function Reference

## A.1 cmfd_init

This subroutine initializes the CMFD module. All work that must be done only once per MCNP run is performed in this subroutine. The following tasks are performed in order:

1. Locate the IDs of the necessary mesh tallies to perform CMFD,
2. analyze the mesh geometry and check for errors,
3. allocate memory for the structures needed during the CMFD analysis.

Since the required memory allocation takes place within `cmfd_init`, it is important that this subroutine be called prior to calling any other subroutines or functions in the CMFD module. The logical variable, `initialized` stores the initialization status of the module. Whenever a call to `cmfd_update` is made a check for `logical == .true.` is made, and in the event that the module is uninitialized, a call to `cmfd_init` is made.

## A.2 cmfd_update

The `cmfd_update` subroutine is called every time it is desired to incorporate new monte carlo tally data into the estimates for the partial currents or multi-group cross sections. Generally, `cmfd_update` would be called each time before calling `cmfd_solve`.

The tally IDs that were located by `cmfd_init` are used to calculate all of the group constants necessary for bulding the FDM/CMFD system. Partial currents are stored in the `j` array after being normalized by the source neutron weight, `sp_norm` and the corresponding mesh surface area.

After storing all partial currents, the other group constants are calculated as per Eq. (4) and flux is stored after normalizing by `sp_norm` and the mesh volume. Removal cross sections are generated using Eq. (5) and $\Sigma_{s12}$ is calculated using Eq. (1).

`cmfd_update` then proceeds to calculate the diffusivities, $\beta$ for each mesh region. In the event of a boundary region, $\beta$ is calculated using the definition of $\beta_s$ from section 2.2.3. Coupling coefficients are then generated using these $\beta$ values.

With all $\tilde{D}$ and $\hat{D}$ defined, the migration matrix is constructed from the balance equation (19). The off-diagonals of the migration matrix are stored in the `coup(dir,x,y,z,grp)` array, which stores the coupling terms between the node at `x,y,z` in the direction `dir` for energy group `grp`. Diagonal terms of the migration matrix are stored in the `diag(x,y,z,grp)` array. To avoid branching statements, entries are calculated for `coup` along the boundaries of the physical problem domain which should technically be zero. These elements are corrected by an additional sweep along each face of the domain to remove these non-zero entries.

## A.3 cmfd_solve

After setting up the equations in `cmfd_update`, the `cmfd_solve` subroutine performs the power method on the system to achieve convergence. This process is carried out by first taking the LU decomposition of the migration matrix with the four outer-most off-diagonals removed via a call to the `cmfd_lu` subroutine. Since this process is contained in the `cmfd_lu` subroutine it is discussed in section A.4. By LU decomposing just

the main diagonal and the two closest off-diagonals, application of the Gauss-Seidel is facilitated, since in effect the outer off-diagonals have been removed to the other side of the equation.

An initial, uniform flux distribution is guess of $k_{eff} = 1$ is assumed and `cmfd_solve` enters the outer iteration. A sweep is performed across all mesh regions to solve for a fission source which will be fixed during the inner iterations. Each energy group is now treated independently, starting with the higher energy groups and moving down. For each group, a scattering source is accumulated using the flux solution from the energy groups above the group of interest. Upscattering is not treated. This scattering source is now added to the fraction of the fission source emitted in the current group and stored in the `src(x,y,z)` array, which is also fixed for the duration of the inner iterations.

Inner iterations are now used to achieve partial convergence given the fixed source provided by the outer iteration. A sweep is performed along the Y and Z directions, and flux is solved along strips in the X direction. Coupling from the north, south, top and bottom, which were previously removed from the migration matrix to facilitate the LU decomposition are added to the right hand side of the system using the flux in these neighboring nodes from the previous inner iteration. This process is repeated for the number of iterations defined by `IDUM(2)`.

Once the inner iterations have been performed for each energy group, the eigenvalue, $k_{eff}$ is calculated as

$$k_{eff} = \frac{\left\langle \phi^l, \phi^l \right\rangle}{\left\langle \phi^l, \phi^{(l-1)} \right\rangle}, \tag{34}$$

where $\phi^l$ is the flux distribution for the $l$-th outer iteration. Estimates of the convergence of $k_{eff}$ and $\phi$ are calculated using

$$\Delta k = |k^l - k^{(l-1)}| \text{and} \tag{35}$$

$$\Delta \phi = |\left\langle \phi, \phi \right\rangle^l - \left\langle \phi, \phi \right\rangle^{(l-1)}|. \tag{36}$$

If these error estimates are below the convergence criteria defined in the module parameters, the subroutine returns. Otherwise, the outer iteration is repeated.

## A.4 cmfd_lu

The `cmfd_lu` subroutine performs an LU decomposition of the block matrices which describe the west-east-coupled strips of mesh elements along the $x$ axis. The matrix being operated upon is the result of subtracting the four outer diagonals of the matrix depicted in Fig. 2. The remaining matrix contains a series of tridiagonal matrices along the diagonal. Each of these tridiagonal blocks correspond to a strip along the $x$-axis at a particular $(y, z)$ position and are treated independently.

The LU decomposition is carried out using an algorithm specific to tridiagonal matrices [1] for efficiency. The resultant matrices, $\mathbf{L}$ and $\mathbf{U}$ preserve the tridiagonal structure of the original matrix, allowing them to be stored as three vectors which define the lower diagonal of $\mathbf{L}$ and the main and upper diagonals of $\mathbf{U}$. The main diagonal of $\mathbf{L}$ is all ones.

## A.5 cmfd_bank

Once a fission source distribution has been obtained from the CMFD solution, `cmfd_bank` adjusts the fission source, `fso_src` to reflect the CMFD solution. The final `fso_src` is constructed from source particles which are already stored in the `fso_src` array. `cmfd_bank` starts by sweeping through the existing fission bank to determine how many source particles exist in each mesh element. The FSD obtained from the CMFD calculation is then used to build a vector of sampling weights corresponding to each source particle in `fso_src`, defined by

$$\text{wgt} = \frac{\psi}{N}, \tag{37}$$

where $\psi$ is the fission source strength in the mesh element of the source particle from the CMFD solution and $N$ is the number of source particles which reside in the mesh element.

Once the weight vector has been constructed, it is passed to `cmfd_sample` which samples particles from `fso_src` based on the weights in the weight vector. `cmfd_sample` returns a list of particle indexes corresponding to entries in the `fso_src` array. These indexes are then used to build `fso_bnk` by copying the associated entries from `fso_src`.

### A.6 cmfd_test

This subroutine is the main point of access to the CMFD module and its functionality. Typically, `cmfd_test` is called from outside of the CMFD module (for instance, between KCODE cycles), and contains function and subroutine calls which control the progression of the CMFD calculation.

## B Source Code

Listing 2: Module Source Code

```
1
2  module cmfd_mod
3
4    use mcnp_params, only: dknd,zero,one,two,three,FSO_XXX,FSO_YYY,FSO_ZZZ
5    use mcnp_global, only: nps,fpi,kcy,nsrck,ikz,fso_src,fso_src_count,fso_bnk,fso_max_count1,
         idum,ntasks,kct
6    use fmesh_mod,   only: nmesh,fm
7    use varcom,   only: kcy
8
9    implicit none
10   private
11
12   logical :: initialized = .false.
13
14   character*40  :: ch40,chtemp
15
16   integer,parameter :: &
17   &  g      = 2,   & ! number of energy groups to treat
18   &  mesh_t  = 14,   & ! mesh id for total interactions
19   &  mesh_nsf   = 24,  & ! mesh id for nu-fission
20   &  mesh_flux  = 4,   & ! mesh id for scalar flux
21   &  mesh_abs   = 34,  & ! mesh id for absorptions
22   &  mesh_f  = 44      ! mesh id for fission
23
24   real(dknd) ::       &
25   &  k_eps = 1.e-5,   & ! convergence criteria for keff
26   &  flux_eps=1.e-4   ! convergence criteria for flux distribution
27
28   integer :: &
29   &  nx,            &  ! Number of nodes in the X direction.
30   &  ny,            &  ! Number of nodes in the Y direction.
31   &  nz,            &  ! Number of nodes in the Z direction.
32   &  n,             &  ! Total number of mesh points.
33   &  mesh_t_id,     &
34   &  mesh_nsf_id,   &
35   &  mesh_flux_id,  &
36   &  mesh_abs_id
37
38   integer,dimension(6) :: j_mesh_id   ! mesh IDs for the current tallies
39
40   integer,allocatable :: nghbr(:,:,:,:) ! map of neighboring regions in each direction
41
42   ! Number of nodes in the Z direction
43   real(dknd) ::     &
44   &  hx,            &  ! width of nodes in X direction.
45   &  hy,            &  ! width of nodes in Y direction.
```

```
46    &  hz,              &  ! width of nodes in Z direction.
47    &  ax,              &  ! surface area in X direction.
48    &  ay,              &  ! surface area in Y direction.
49    &  az,              &  ! surface area in Z direction.
50    &  v,               &  ! volume of mesh cells.
51    &  alb_xl=1.e30,    &
52    &  alb_xr=1.e30,    &
53    &  alb_yl=1.e30,    &
54    &  alb_yr=1.e30,    &
55    &  alb_zl=1.e30,    &
56    &  alb_zr=1.e30,    &
57    &  keff
58
59    real(dknd),dimension(3) :: mesh_orig
60
61    real(dknd),allocatable :: &
62    &  flux(:,:,:,:),          & ! flux in each node (x,y,z)/group.
63    &  flux_old(:,:,:,:),      & ! flux from previous iteration.
64    &  flux_old_out(:,:,:,:), &
65    &  d(:,:,:,:),             & ! Diffusion coefficient in each node/group.
66    &  nu_sf(:,:,:,:),         & ! nu-fission "               ".
67    &  j(:,:,:,:,:),       & ! currents in each direction/face/energy. Starts at the zero-th
          face.
68    &  sigt(:,:,:,:),          & ! total cross section.
69    &  siga(:,:,:,:),          & ! absorption cross section.
70    &  sigscat(:,:,:,:,:),     & ! scattering matrix (g',g,x,y,z). Groups come first for cache
          eff.
71    &  sigr(:,:,:,:),          & ! Removal cross section.
72    &  d_tilde_x(:,:,:,:),     & !
73    &  d_tilde_y(:,:,:,:),     &
74    &  d_tilde_z(:,:,:,:),     &
75    &  beta_x(:,:,:,:),        & ! Relative diffusivity in X direction.
76    &  beta_y(:,:,:,:),        & ! Relative diffusivity in Y direction.
77    &  beta_z(:,:,:,:),        & ! Relative diffusivity in Z direction.
78    &  diag(:,:,:,:),          & ! diagonal vector of the migration matrix
79    &  lu_ud(:,:,:,:),         & !
80    &  lu_uu(:,:,:,:),         &
81    &  lu_ll(:,:,:,:),         &
82    &  psi(:,:,:),             &
83    &  psi_old(:,:,:),         &
84    &  lu_b(:),                &
85    &  lu_y(:),                &
86    &  src(:,:,:),             &
87    &  chi(:,:,:,:),       &
88    &  coup_x(:,:,:,:),        & !
89    &  coup_y(:,:,:,:),        &
90    &  coup_z(:,:,:,:),        &
91    &  d_hat_x(:,:,:,:),       &
92    &  d_hat_y(:,:,:,:),       &
93    &  d_hat_z(:,:,:,:),       &
94    &  coup(:,:,:,:,:),        &
95    &  fso_new(:,:,:,:),       &
96    &  sample_wgt(:),          & ! list of weights provided to sampling routine
97    &  flux_avg(:),            &
98    &  egrp(:),                &
99    &  scat_tal(:,:,:,:,:,:), &
100   &  diag2(:,:,:,:),         &
101   &  fsrc_pop(:)
102
103
104   integer,allocatable ::     &
105   &  fsrc_pos(:),            &
106   &  fsrc_ind(:)
107
108   public :: cmfd_test
```

```
109
110
111   contains
112 ! ============================================================================
113   ! initialize the cmfd module with problem gemometry, etc.
114   subroutine cmfd_init()
115     integer :: i,x,y,z,k
116
117     open(unit=999,file='junk')
118
119     ! locate the IDs of the mesh current tallies.
120       do i=1,nmesh
121       k = mod(fm(i)%id,100)
122       write(*,*)"id: ", fm(i)%id,k
123       ! write(*,*)"fm info:",fm(i)%
124       select case(k)
125         case(11)
126           j_mesh_id(1) = i
127         case(21)
128           j_mesh_id(2) = i
129         case(31)
130           j_mesh_id(3) = i
131         case(41)
132           j_mesh_id(4) = i
133         case(51)
134           j_mesh_id(5) = i
135         case(61)
136           j_mesh_id(6) = i
137       end select
138     end do
139     ! locate the other tallies neccesary for calculating MG cross sections
140     do i=1,nmesh
141       select case(fm(i)%id)
142         case(mesh_t)
143           mesh_t_id = i
144         case(mesh_nsf)
145           mesh_nsf_id  = i
146         case(mesh_flux)
147           mesh_flux_id = i
148         case(mesh_abs)
149           mesh_abs_id  = i
150         end select
151     end do
152
153
154     ! TODO do error checking on the tally specs to make sure they all match
155     ! look at the tallies to get the geom info
156     nx = fm(j_mesh_id(1))%nxrb - 3
157     ny = fm(j_mesh_id(1))%nyzb - 3
158     nz = fm(j_mesh_id(1))%nztb - 3
159     !Set up geometry
160     n = nx*ny*nz
161     hx = fm(j_mesh_id(1))%xrbin(2)-fm(j_mesh_id(1))%xrbin(1)
162     hy = fm(j_mesh_id(1))%yzbin(2)-fm(j_mesh_id(1))%yzbin(1)
163     hz = fm(j_mesh_id(1))%ztbin(2)-fm(j_mesh_id(1))%ztbin(1)
164     ax = hy*hz
165     ay = hx*hz
166     az = hx*hy
167     v = hx*hy*hz
168
169     ! find the origin of the ACTIVE region of the mesh
170     mesh_orig(1) = fm(mesh_flux_id)%xrbin(2)
171     mesh_orig(2) = fm(mesh_flux_id)%yzbin(2)
172     mesh_orig(3) = fm(mesh_flux_id)%ztbin(2)
173
```

```
174       write(*,*)"fso_src_count:",fso_src_count
175
176       write(*,*)"mesh diemensions"
177       write(*,*)"meshes: ", nx,ny,nz
178       write(*,*)"mesh widths: ",hx,hy,hz
179       write(*,*)"mesh volume: ",v
180       write(*,*)"more ids: ",mesh_t_id
181       ! allocate memory
182       allocate( flux(nx,ny,nz,g) )
183       allocate( sigt(nx,ny,nz,g) )
184       allocate( d(nx,ny,nz,g) )
185       allocate( nu_sf(nx,ny,nz,g) )
186       allocate( j(6,0:nx,0:ny,0:nz,g) )
187       allocate( siga(nx,ny,nz,g) )
188       allocate( sigr(nx,ny,nz,g) )
189       allocate( sigscat(g,g,nx,ny,nz) )
190       allocate( beta_x(0:(nx+1),ny,nz,g) )
191       allocate( beta_y(nx,0:(ny+1),nz,g) )
192       allocate( beta_z(nx,ny,0:(nz+1),g) )
193       allocate( d_tilde_x(0:nx,ny,nz,g) )
194       allocate( d_tilde_y(nx,0:ny,nz,g) )
195       allocate( d_tilde_z(nx,ny,0:nz,g) )
196       allocate( lu_ll(nx,ny,nz,g) )
197       allocate( lu_ud(nx,ny,nz,g) )
198       allocate( lu_uu(nx,ny,nz,g) )
199       allocate( flux_old(nx,ny,nz,g) )
200       allocate( flux_old_out(nx,ny,nz,g) )
201       allocate( psi(nx,ny,nz) )
202       allocate( psi_old(nx,ny,nz) )
203       allocate( lu_b(nx) )
204       allocate( lu_y(nx) )
205       allocate( src(nx,ny,nz) )
206       allocate( chi(nx,ny,nz,g) )
207       allocate( diag(nx,ny,nz,g) )
208       allocate( coup(6,nx,ny,nz,g) )
209       allocate( d_hat_x(0:nx,ny,nz,g) )
210       allocate( d_hat_y(nx,0:ny,nz,g) )
211       allocate( d_hat_z(nx,ny,0:nz,g) )
212       allocate( fsrc_pop(n) )
213       allocate( sample_wgt(fso_max_count1) )
214       allocate( fsrc_pos(fso_max_count1) )
215       allocate( fsrc_ind(fso_max_count1) )
216       allocate( flux_avg(g) )
217       allocate( egrp(g) )
218       allocate( scat_tal(0:ntasks-1,g,g,nx,ny,nz) )
219       allocate( diag2(nx,ny,nz,g) )
220
221
222       ! zero out some stuff that might not get initialized
223       sigscat = 0.0
224       j = 0.0
225
226       write(*,*)"kct",kct
227
228       write(*,*)"tally ids:"
229       write(*,*) j_mesh_id
230
231       ! flag the module as initialized
232       initialized = .true.
233       call cmfd_update
234
235
236     end subroutine cmfd_init
237   ! ============================================================================
238     ! update material properties for each node
```

```
  subroutine cmfd_update
    integer :: node,grp,x,y,z,ig,i
    integer :: xp,yp,zp
    real(dknd) :: sp_norm,tempr1,tempr2,phi_l,phi_r,avg_flux_1,avg_flux_2
    real(dknd) :: flux_tal,siga_tal,sigt_tal,nsf_tal

    ! collect tally data from nodes
!   call fmesh_msgcon

    ! tally normalization based on source histories
    sp_norm = (kcy-ikz)*nsrck
    do grp=1,g
      ! reverse the group order to follow high->low convention
      ig = g-grp+1
      ! grab the surface currents from the mesh tally. this gets kind of goofy, so
      ! we will do it one direction at a time.
      ! X direction
      do x=0,nx
        do y=1,ny
          do z=1,nz
            j(1,x,y,z,ig) = fm(j_mesh_id(1))%fmarry(x+1,y+1,z+1,grp,1)/(sp_norm*ax) ! x+
            j(2,x,y,z,ig) = fm(j_mesh_id(2))%fmarry(x+2,y+1,z+1,grp,1)/(sp_norm*ax) ! x-
          end do ! z
        end do ! y
      end do ! x
      do y=0,ny
        do x=1,nx
          do z=1,nz
            j(3,x,y,z,ig) = fm(j_mesh_id(3))%fmarry(x+1,y+1,z+1,grp,1)/(sp_norm*ay) ! y+
            j(4,x,y,z,ig) = fm(j_mesh_id(4))%fmarry(x+1,y+2,z+1,grp,1)/(sp_norm*ay) ! y-
          end do ! z
        end do ! x
      end do ! y
      do z=0,nz
        do x=1,nx
          do y=1,ny
            j(5,x,y,z,ig) = fm(j_mesh_id(5))%fmarry(x+1,y+1,z+1,grp,1)/(sp_norm*az) ! z+
            j(6,x,y,z,ig) = fm(j_mesh_id(6))%fmarry(x+1,y+1,z+2,grp,1)/(sp_norm*az) ! z-
          end do ! y
        end do ! x
      end do ! z
      ! calculate cross sections and fetch important data for each node/group
      do z=1,nz
        do y=1,ny
          do x=1,nx
            xp = x+1
            yp = y+1
            zp = z+1
            ! get the raw tally values for the current region/group
            flux_tal = fm(mesh_flux_id)%fmarry(xp,yp,zp,grp,1)
            siga_tal = fm(mesh_abs_id)%fmarry(xp,yp,zp,grp,1)
            sigt_tal = fm(mesh_t_id)%fmarry(xp,yp,zp,grp,1)
            nsf_tal  = fm(mesh_nsf_id)%fmarry(xp,yp,zp,grp,1)
            flux(x,y,z,ig)  = fm(mesh_flux_id)%fmarry(xp,yp,zp,grp,1)/(sp_norm*v) ! Flux
            ! nu-fission
            nu_sf(x,y,z,ig) = nsf_tal/flux_tal
            ! total macroscopic cross section
            sigt(x,y,z,ig)  = sigt_tal/flux_tal
            ! absorption cross section
            siga(x,y,z,ig)  = siga_tal/flux_tal
            ! calculate diffusion coefficient
            d(x,y,z,ig) = one/(three*sigt(x,y,z,ig))

            chi(x,y,z,1) = 1.0
            chi(x,y,z,2) = 0.0
```

```
          end do ! x
        end do ! y
      end do ! z
    end do ! group
    ! sweep back through to calculate scattering (1->2) and removal cross sections
    do z=1,nz
      do y=1,ny
        do x=1,nx
          xp = x+1
          yp = y+1
          zp = z+1
          siga_tal = fm(mesh_abs_id)%fmarry(xp,yp,zp,1,1)
          flux_tal = fm(mesh_flux_id)%fmarry(xp,yp,zp,2,1)
          ! Calculate the in-scattering cross section (Sig_s12).
          sp_norm = 0
          tempr1 = (siga_tal + sp_norm*(cmfd_jout(x,y,z,2) - cmfd_jin(x,y,z,2)))/flux_tal

          sigscat(1,2,x,y,z) = tempr1
!         if(sigscat(1,2,x,y,z)<0.0) then
!           sigscat(1,2,x,y,z) = 0.0
!         end if
          ! calculate the removal cross section
          sigr(x,y,z,1) = siga(x,y,z,1) + sigscat(1,2,x,y,z)
          sigr(x,y,z,2) = siga(x,y,z,2)
        end do ! x
      end do ! y
    end do ! z

    ! store the group boundaries
    do grp=2,g
      egrp(grp) = fm(mesh_flux_id)%enbin(grp)
    end do

    ! print out the jin and jout
    open(file="plane_jin1" ,unit=500)
    open(file="plane_jout1",unit=501)
    open(file="plane_jin2",unit=502)
    open(file="plane_jout2",unit=503)
    open(file="plane_jl1",unit=504)
    open(file="plane_jl2",unit=505)

    do y=1,ny
      do x=1,nx
        write(500,*)x,y,cmfd_jin(x,y,nz/2,1)
        write(501,*)x,y,cmfd_jout(x,y,nz/2,1)
        write(502,*)x,y,cmfd_jin(x,y,nz/2,2)
        write(503,*)x,y,cmfd_jout(x,y,nz/2,2)
        write(504,*)x,y,(cmfd_jout(x,y,nz/2,1)-cmfd_jin(x,y,nz/2,1))
        write(505,*)x,y,(cmfd_jout(x,y,nz/2,2)-cmfd_jin(x,y,nz/2,2))
      end do
      write(500,*)
      write(501,*)
      write(502,*)
      write(503,*)
      write(504,*)
      write(505,*)
    end do

    close(500)
    close(501)
    close(502)
    close(503)
    close(504)
    close(505)
```

```
369       do grp=1,g
370         flux_avg(grp) = SUM(flux(:,:,:,grp))/REAL(COUNT(flux(:,:,:,grp).ge.0),dknd)
371       end do
372
373       ! check for zero flux
374       do grp=1,g
375         do z=1,nz
376           do y=1,ny
377             do x=1,nx
378               if (flux(x,y,z,grp)==0) then
379                 write(*,*)"Zero flux in region:",x,y,z,grp
380                 ! put in some placeholder numbers to keep the solver from crashing
381                 ! use an average value of flux
382                 flux(x,y,z,grp) = flux_avg(grp)
383                 ! set D to be the average of hx,hy,hz to get a beta of one-ish
384                 d(x,y,z,grp) = (hx+hy+hz)/three
385                 ! zero out everything else
386                 siga(x,y,z,grp) = 0
387                 sigr(x,y,z,grp) = 0
388                 sigscat(:,:,x,y,z) = 0
389                 nu_sf(x,y,z,grp) = 0
390               end if
391             end do
392           end do
393         end do
394       end do
395
396       ! hard code some cross sections for testing
397   !   d(:,:,:,1) = 0.1666667
398   !   d(:,:,:,2) = 0.1111111
399   !   sigr(:,:,:,1) = 1.5
400   !   sigr(:,:,:,2) = 2.0
401   !   nu_sf(:,:,:,1) = 0.375
402   !   nu_sf(:,:,:,2) = 4.5
403   !   sigscat = 0.0
404   !   sigscat(1,2,:,:,:) = 0.5
405
406
407       diag = 0.0
408       diag2 = 0.0
409
410       ! calculate relative diffusivity, beta
411       do grp=1,g
412         ! interior regions
413         do z=1,nz
414           do y=1,ny
415             do x=1,nx
416               beta_x(x,y,z,grp) = d(x,y,z,grp)/hx
417               beta_y(x,y,z,grp) = d(x,y,z,grp)/hy
418               beta_z(x,y,z,grp) = d(x,y,z,grp)/hz
419             end do ! x
420           end do ! y
421         end do ! z
422         ! define the extremities using albedo.
423
424         ! X direction (east/west faces)
425         do z=1,nz
426           do y=1,ny
427             beta_x(0,y,z,grp)    = 0.5*ABS(j(1,0,y,z,grp)-j(2,0,y,z,grp))/flux(1,y,z,grp)
428             beta_x(nx+1,y,z,grp) = 0.5*ABS(j(2,nx+1,y,z,grp)-j(1,nx+1,y,z,grp))/flux(nx,y,z,
                    grp)
429   !         beta_x(0,y,z,grp) = alb_xl*0.5
430   !         beta_x(nx+1,y,z,grp) = alb_xr*0.5
431           end do ! y
432         end do ! z
```

```
433       ! Y direction (north/south faces)
434       do z=1,nz
435         do x=1,nx
436           beta_y(x,0,z,grp)    = 0.5*ABS(j(3,x,0,z,grp)-j(4,x,0,z,grp))/flux(x,1,z,grp)
437           beta_y(x,ny+1,z,grp) = 0.5*ABS(j(4,x,ny+1,z,grp)-j(3,x,ny+1,z,grp))/flux(x,ny,z,
                  grp)
438   !       beta_y(x,0,z,grp) = alb_yl*0.5
439   !       beta_y(x,ny+1,z,grp) = alb_yr*0.5
440         end do ! x
441       end do ! z
442       ! Z direction (top/bottom faces)
443       do y=1,ny
444         do x=1,nx
445           beta_z(x,y,0,grp)    = 0.5*ABS(j(5,x,y,0,grp)-j(6,x,y,0,grp))/flux(x,y,1,grp)
446           beta_z(x,y,nz+1,grp) = 0.5*ABS(j(6,x,y,nz+1,grp)-j(5,x,y,nz+1,grp))/flux(x,y,nz,
                  grp)
447   !       beta_z(x,y,0,grp) = alb_zl*0.5
448   !       beta_z(x,y,nz+1,grp) = alb_zr*0.5
449         end do ! x
450       end do ! y
451     end do ! grp
452
453     ! Calculate d_tilde and d_hat.
454     do grp=1,g
455       ! X direction
456       do z=1,nz
457         do y=1,ny
458           do x=0,nx
459             d_tilde_x(x,y,z,grp) = two*beta_x(x,y,z,grp)*beta_x(x+1,y,z,grp)&
460             &  /(beta_x(x,y,z,grp)+beta_x(x+1,y,z,grp))
461             ! define flux to the left and right of the surface
462             if (x==0) then
463               phi_l = 0.0
464             else
465               phi_l = flux(x,y,z,grp)
466             end if
467             if (x==nx) then
468               phi_r = 0.0
469             else
470               phi_r = flux(x+1,y,z,grp)
471             end if
472             ! d_hat                    x+              x-
473             d_hat_x(x,y,z,grp) = ((j(1,x,y,z,grp)-j(2,x,y,z,grp))+d_tilde_x(x,y,z,grp)&
474             &  *(phi_r-phi_l))/(phi_r+phi_l)
475           end do ! x
476         end do ! y
477       end do ! z
478       ! Y direction
479       do z=1,nz
480         do y=0,ny
481           do x=1,nx
482             d_tilde_y(x,y,z,grp) = two*beta_y(x,y,z,grp)*beta_y(x,y+1,z,grp)&
483             &  /(beta_y(x,y,z,grp)+beta_y(x,y+1,z,grp))
484             ! define flux to the left and right of the surface
485             if (y==0) then
486               phi_l = 0.0
487             else
488               phi_l = flux(x,y,z,grp)
489             end if
490             if (y==ny) then
491               phi_r = 0.0
492             else
493               phi_r = flux(x,y+1,z,grp)
494             end if
495             ! d_hat
```

```
496             d_hat_y(x,y,z,grp) = ((j(3,x,y,z,grp)-j(4,x,y,z,grp))+d_tilde_y(x,y,z,grp)&
497             &  *(phi_r-phi_l))/(phi_r+phi_l)
498           end do ! x
499         end do ! y
500       end do ! z
501       ! Z direction
502       do z=0,nz
503         do y=1,ny
504           do x=1,nx
505             d_tilde_z(x,y,z,grp) = two*beta_z(x,y,z,grp)*beta_z(x,y,z+1,grp)&
506             &  /(beta_z(x,y,z,grp)+beta_z(x,y,z+1,grp))
507             ! define flux to the left and right of the surface
508             if (z==0) then
509               phi_l = 0.0
510             else
511               phi_l = flux(x,y,z,grp)
512             end if
513             if (z==nz) then
514               phi_r = 0.0
515             else
516               phi_r = flux(x,y,z+1,grp)
517             end if
518             ! d_hat
519             d_hat_z(x,y,z,grp) = ((j(5,x,y,z,grp)-j(6,x,y,z,grp))+d_tilde_z(x,y,z,grp)&
520             &  *(phi_r-phi_l))/(phi_r+phi_l)
521           end do ! x
522         end do ! y
523       end do ! z
524
525       if (idum(4)/=0) then
526         write(*,*)"Turning off CMFD."
527         d_hat_x = 0
528         d_hat_y = 0
529         d_hat_z = 0
530       end if
531
532       write(chtemp,*)kcy
533       chtemp = adjustl(chtemp)
534       write(ch40,*)'plane_dhat',trim(chtemp)
535       open(223,file=ch40)
536       z = nz/2
537       do y=1,ny
538         do x=1,nx
539           write(223,*)x,y,d_hat_x(x,y,z,1)
540         end do
541         write(223,*)
542       end do
543       close(223)
544
545       do z=1,nz
546         do y=1,ny
547           do x=1,nx
548             coup(1,x,y,z,grp) = ax * (-d_tilde_x(x,y,z,grp)   + d_hat_x(x,y,z,grp))   ! east
549             coup(2,x,y,z,grp) = ax * (-d_tilde_x(x-1,y,z,grp) - d_hat_x(x-1,y,z,grp)) ! west
550             coup(3,x,y,z,grp) = ay * (-d_tilde_y(x,y,z,grp)   + d_hat_y(x,y,z,grp))   !
                    south
551             coup(4,x,y,z,grp) = ay * (-d_tilde_y(x,y-1,z,grp) - d_hat_y(x,y-1,z,grp)) !
                    north
552             coup(5,x,y,z,grp) = az * (-d_tilde_z(x,y,z,grp)   + d_hat_z(x,y,z,grp))   ! down
553             coup(6,x,y,z,grp) = az * (-d_tilde_z(x,y,z-1,grp) - d_hat_z(x,y,z-1,grp)) ! up
554           end do
555         end do
556       end do
557
558
```

```
559       ! Form the diagonal of the migration matrix
560       do z=1,nz
561         do y=1,ny
562           do x=1,nx
563             diag(x,y,z,grp) = &
564             &  ax*(d_tilde_x(x,y,z,grp)+d_tilde_x(x-1,y,z,grp)+d_hat_x(x,y,z,grp)-d_hat_x(x
                    -1,y,z,grp)) + &
565             &  ay*(d_tilde_y(x,y,z,grp)+d_tilde_y(x,y-1,z,grp)+d_hat_y(x,y,z,grp)-d_hat_y(x,
                    y-1,z,grp)) + &
566             &  az*(d_tilde_z(x,y,z,grp)+d_tilde_z(x,y,z-1,grp)+d_hat_z(x,y,z,grp)-d_hat_z(x,
                    y,z-1,grp))
567             do i=1,6
568               diag2(x,y,z,grp) = diag2(x,y,z,grp)-coup(i,x,y,z,grp)
569             end do
570             diag(x,y,z,grp) = diag(x,y,z,grp) + sigr(x,y,z,grp)*v
571             diag2(x,y,z,grp) = diag2(x,y,z,grp) + sigr(x,y,z,grp)*v
572           end do ! x
573         end do ! y
574       end do ! z
575
576       ! calculate coupling coefficients using d-hat,d-tilde and area
577       ! clean up the boundary
578       ! east/west face
579       do z=1,nz
580         do y=1,ny
581           coup(1,nx,y,z,grp) = 0.0
582           coup(2,1,y,z,grp)  = 0.0
583         end do
584       end do
585       ! north/south faces
586       do z=1,nz
587         do x=1,nx
588           coup(3,x,ny,z,grp) = 0.0
589           coup(4,x,1,z,grp)  = 0.0
590         end do
591       end do
592       ! top/bottom faces
593       do y=1,ny
594         do x=1,nx
595           coup(5,x,y,nz,grp) = 0.0
596           coup(6,x,y,1,grp)  = 0.0
597         end do
598       end do
599     end do ! grp
600
601     ch40 = 'plane_f1'
602     call print_plane_z(2,1,flux,ch40)
603
604     open(unit=998,file='plane_s12')
605     do x=1,nx
606       do y=1,ny
607         write(998,'(i4,2x,i4,2x,e12.5)')x,y,sigscat(1,2,x,y,2)
608       end do
609       write(998,*)
610     end do
611     close(998)
612     ch40 = 'plane_r1'
613     call print_plane_z(2,1,sigr,ch40)
614     ch40 = 'plane_r2'
615     call print_plane_z(2,2,sigr,ch40)
616
617     ch40 = 'plane_t1'
618     call print_plane_z(2,1,sigt,ch40)
619     ch40 = 'plane_nsf1'
620     call print_plane_z(2,1,nu_sf,ch40)
```

```
621       ch40 = 'plane_a1'
622       call print_plane_z(2,1,siga,ch40)
623       ch40 = 'plane_d1'
624       call print_plane_z(2,1,d,ch40)
625       ch40 = 'plane_f2'
626       call print_plane_z(2,2,flux,ch40)
627       ch40 = 'plane_t2'
628       call print_plane_z(2,2,sigt,ch40)
629       ch40 = 'plane_nsf2'
630       call print_plane_z(2,2,nu_sf,ch40)
631       ch40 = 'plane_a2'
632       call print_plane_z(2,2,siga,ch40)
633       ch40 = 'plane_d2'
634       call print_plane_z(2,2,d,ch40)
635
636       open(unit=998,file='stuff')
637       write(998,*)"two: ",two
638       write(998,*)"three: ",three
639       write(998,*)
640       write(998,*)"coup:"
641       do x=1,nx
642         write(998,*)coup(:,x,ny/2,nz/2,1)
643         write(998,*)
644       end do
645 !     write(998,*)"flux"
646 !     write(998,*)flux
647 !     write(998,*)
648       write(998,*)
649       write(998,*)"diag:"
650       write(998,*)diag
651       write(998,*)"diag2:"
652       write(998,*)diag2
653       write(998,*)
654       write(998,*)"d_tilde_x"
655       write(998,*)d_tilde_x
656       close(998)
657
658       open(unit=998,file="migration")
659       write(998,*)"#   west east north south up down diag"
660       do grp=1,g
661         do z=1,nz
662           do y=1,ny
663             do x=1,nx
664
665               if(x>1)then
666                 write(998,'(1p,e12.5,1x)',advance='no')coup(2,x,y,z,grp)
667               else
668                 write(998,'(1p,e12.5,1x)',advance='no')0.
669               end if
670
671               if(x<nx)then
672                 write(998,'(1p,e12.5,1x)',advance='no')coup(1,x,y,z,grp)
673               else
674                 write(998,'(1p,e12.5,1x)',advance='no')0.
675               end if
676               if(y>1)then
677                 write(998,'(1p,e12.5,1x)',advance='no')coup(4,x,y,z,grp)
678               else
679                 write(998,'(1p,e12.5,1x)',advance='no')0.
680               end if
681               if(y<ny)then
682                 write(998,'(1p,e12.5,1x)',advance='no')coup(3,x,y,z,grp)
683               else
684                 write(998,'(1p,e12.5,1x)',advance='no')0.
685               end if
```

```
            if(z>1)then
              write(998,'(1p,e12.5,1x)',advance='no')coup(6,x,y,z,grp)
            else
              write(998,'(1p,e12.5,1x)',advance='no')0.
            end if
            if(z<nz)then
              write(998,'(1p,e12.5,1x)',advance='no')coup(5,x,y,z,grp)
            else
              write(998,'(1p,e12.5,1x)',advance='no')0.
            end if

            write(998,'(1p,e12.5,1x)')diag(x,y,z,grp)

          end do
        end do
      end do
    end do
    close(998)

  end subroutine cmfd_update
! ======================================================================
  ! do some stuff to see if we are working
  subroutine cmfd_test(kin)
    real(dknd) :: kin
    integer :: i,fsrc_tot,x,y,z

    keff = kin

    if(idum(1)==0) then
      return
    end if

    if(initialized.eq..false.) then
      ! initialize CMFD module
      write(*,*)"Initializing CMFD module."
      call cmfd_init
    else
      if (kcy.eq.idum(1)) then
        ! do an update
        call cmfd_update
        call cmfd_solve(idum(3),idum(2)) ! outer_it,inner_it
        call cmfd_bank
      end if
      if (kcy==idum(1)+1) then
        call cmfd_update
        ! print flux
        ch40 = 'plane_fd1'
        call print_plane_z(nz/2,1,flux,ch40)
        ch40 = 'plane_fd2'
        call print_plane_z(nz/2,2,flux,ch40)
      end if
      if (kcy==kct-1) then
        ! take the residual of the fission source to the converged FSD
        call cmfd_pop
        fsrc_tot = SUM(fsrc_pop)
        ! normalize psi
        psi = psi/SUM(psi)
        open(unit=444,file='plane_psierr')
        z = nz/2
        do y=1,ny
          do x=1,nx
            i = nx*ny*(z-1)+nx*(y-1)+x
            write(444,*)x,y,( ((fsrc_pop(i)/fsrc_tot) - psi(x,y,z))/psi(x,y,z) )
          end do
          write(444,*)
```

```
751         end do
752         close(444)
753       end if
754     end if
755   end subroutine cmfd_test
756 ! ==========================================================================
757   subroutine cmfd_normalize
758
759   end subroutine cmfd_normalize
760 ! ==========================================================================
761 ! Returns the outgoing current for the mesh region at x,y,z for neutrons in
762 ! group grp.
763   function cmfd_jout(x,y,z,grp)
764     integer :: x,y,z,grp
765     real  :: cmfd_jout
766
767     cmfd_jout = 0.0 ! initialize output variable
768     ! add contributions for each face
769     cmfd_jout =             j(2,x-1,y,z,grp)*ax ! x- direction at west face
770     cmfd_jout = cmfd_jout + j(1,x,y,z,grp)*ax   ! x+ direction at east face
771     cmfd_jout = cmfd_jout + j(4,x,y-1,z,grp)*ay ! y- direction at north face
772     cmfd_jout = cmfd_jout + j(3,x,y,z,grp)*ay   ! y+ direction at south face
773     cmfd_jout = cmfd_jout + j(6,x,y,z-1,grp)*az ! z- direction at top face
774     cmfd_jout = cmfd_jout + j(5,x,y,z,grp)*az   ! z+ direction at bottom face
775
776   end function cmfd_jout
777 ! ==========================================================================
778 ! Returns the incomming current for the mesh region at x,y,z for neutrons in
779 ! group grp.
780   function cmfd_jin(x,y,z,grp)
781     integer :: x,y,z,grp
782     real  :: cmfd_jin
783
784     cmfd_jin = 0.0 ! initialize output variable
785     ! add contributions for each face
786     cmfd_jin =            j(1,x-1,y,z,grp)*ax ! x+ direction at west face
787     cmfd_jin = cmfd_jin + j(2,x,y,z,grp)*ax   ! x- direction at east face
788     cmfd_jin = cmfd_jin + j(3,x,y-1,z,grp)*ay ! y+ direction at north face
789     cmfd_jin = cmfd_jin + j(4,x,y,z,grp)*ay   ! y- direction at south face
790     cmfd_jin = cmfd_jin + j(5,x,y,z-1,grp)*az ! z+ direction at top face
791     cmfd_jin = cmfd_jin + j(6,x,y,z,grp)*az   ! z- direction at bottom face
792
793   end function cmfd_jin
794 ! ==========================================================================
795   subroutine print_plane_z(z,grp,arry,filename)
796     integer :: x,y,z,grp
797     character*40 :: filename
798     real(dknd),allocatable :: arry(:,:,:,:)
799     open(file=filename,unit=888)
800     do x=1,nx
801       do y=1,ny
802         write(888,'(i4,2x,i4,2x,e12.5)')x,y,arry(x,y,z,grp)
803       end do ! y
804       write(888,*)
805     end do ! z
806     close(888)
807   end subroutine
808 ! ==========================================================================
809   subroutine cmfd_solve(outer_it,inner_it)
810     integer :: x,y,z,grp,outer_it,inner_it,i
811     integer :: outer,inner
812     real(dknd) :: lambda,k_old,tempd1,k_err,flux_err,flux_dot,flux_dot_old
813
814     ! we should have updated cross sections. lu decompose:
815     call cmfd_lu
```

```
816
817     ! start with uniform flux
818     flux = 1.0
819     flux_old = 1.0
820     flux_old_out = 1.0
821     keff = 1.0
822
823     ! open the convergence file, write header
824     open(111,file='converge')
825     write(111,*)"# iteration     k    k_err   flux_err"
826
827     do outer=1,outer_it
828       ! store old flux distribution
829       flux_old_out = flux
830       lambda = 1.0/keff
831
832       ! calculate fission source (psi) at each node
833       psi_old = psi
834       do z=1,nz
835         do y=1,ny
836           do x=1,nx
837             psi(x,y,z) = 0.0
838             do grp=1,g
839               psi(x,y,z) = psi(x,y,z)+nu_sf(x,y,z,grp)*flux(x,y,z,grp)*v
840             end do
841           end do ! x
842         end do ! y
843       end do ! z
844       ! begin the group major portion
845       do grp=1,g
846         ! set up the source
847         do z=1,nz
848           do y=1,ny
849             do x=1,nx
850               src(x,y,z) = psi(x,y,z) * chi(x,y,z,grp) * lambda
851               ! consider scattering (assuming downscatter only)
852               ! TODO: implement upscattering?
853               tempd1 = 0.0
854               do i=1,grp-1
855                 tempd1 = tempd1 + sigscat(i,grp,x,y,z)*flux_old(x,y,z,i)
856               end do ! scattering group
857               src(x,y,z) = src(x,y,z) + tempd1*v ! add scattering to source
858             end do ! x
859           end do ! y
860         end do ! z
861
862         ! now solve this `one group' problem with Gauss-Seidel
863         do inner=1,inner_it
864           flux_old(:,:,:,grp) = flux(:,:,:,grp)
865           do z=1,nz
866             do y=1,ny
867               ! transfer appropriate values from src(:) to lu_b(:),
868               ! then add N,S,U,D coupling to lu_b(:)
869               do x=1,nx
870                 lu_b(x) = src(x,y,z)
871                 ! Add in external sources as needed. This branching is kinda gross,
872                 ! but oh well.
873                 if (y>1) then
874                   lu_b(x) = lu_b(x) - flux(x,y-1,z,grp)*coup(4,x,y,z,grp)   ! north
875                 end if
876                 if (y<ny) then
877                   lu_b(x) = lu_b(x) - flux(x,y+1,z,grp)*coup(3,x,y,z,grp)   ! south
878                 end if
879                 if (z>1) then
880                   lu_b(x) = lu_b(x) - flux(x,y,z-1,grp)*coup(6,x,y,z,grp) ! up
```

```
881                 end if
882                 if (z<nz) then
883                   lu_b(x) = lu_b(x) - flux(x,y,z+1,grp)*coup(5,x,y,z,grp)   ! down
884                 end if
885               end do ! x
886               ! the RHS is now set up. begin forward-backward substitution
887               ! forward substitution
888               lu_y(1) = lu_b(1)
889               do x=2,nx
890                 lu_y(x) = lu_b(x) - lu_ll(x,y,z,grp)*lu_y(x-1)
891               end do
892               ! Do backwards substitution
893               flux(nx,y,z,grp) = lu_y(nx)/lu_ud(nx,y,z,grp)
894               if (flux(nx,y,z,grp)<0.0) then
895                 flux(nx,y,z,grp) = 0.0
896               end if
897               do x=nx-1,1,-1
898                 !                          east
899                 flux(x,y,z,grp) = (lu_y(x)-coup(1,x,y,z,grp)*flux(x+1,y,z,grp))/lu_ud(x,y,z,
                      grp)
900                 if (flux(x,y,z,grp)<0.0) then
901                   flux(x,y,z,grp) = 0.0
902                 end if
903               end do
904             end do ! y (gauss-seidel beams)
905           end do ! z (gauss-seidel planes)
906
907           ch40 = 'plane_fs1'
908           call print_plane_z(nz/2,1,flux,ch40)
909           ch40 = 'plane_fs2'
910           call print_plane_z(nz/2,2,flux,ch40)
911
912         end do ! inner iteration
913       end do ! major group
914       ! solve for k and check for convergence.
915       k_old = keff
916       tempd1 = 0.0
917       flux_dot_old = flux_dot
918       flux_dot  = 0.0
919       ! loop through all the nodes and take <flux,flux> and <flux,flux_old>
920       do grp=1,g
921         do z=1,nz
922           do y=1,ny
923             do x=1,nx
924               tempd1 = tempd1 + flux(x,y,z,grp)*flux(x,y,z,grp)
925               flux_dot = flux_dot + flux(x,y,z,grp)*flux_old_out(x,y,z,grp)
926             end do
927           end do
928         end do
929       end do
930       keff = k_old*(tempd1/flux_dot)
931       flux_err = flux_dot-flux_dot_old
932 !     write(*,*)"k: ",keff
933       k_err = keff-k_old
934       ! print out the convergence info
935       write(111,'(1p,i6,1x,e12.5,1x,e12.5,1x,e12.5)')outer,keff,k_err,flux_err
936       if (mod(outer,10)==0) then
937         write(*,*)outer
938       end if
939       ! convergence?
940         if(ABS(k_err)<k_eps.and.ABS(flux_err)<flux_eps) then
941         ! converged!
942         write(*,*)"CMFD converged in ",outer," iterations! :-D"
943         write(*,*)"k=",keff
944         ch40 = 'plane_fs1'
```

```
945         call print_plane_z(nz/2,1,flux,ch40)
946         ch40 = 'plane_fs2'
947         call print_plane_z(nz/2,2,flux,ch40)
948 
949         ! plot the fission source
950         open(unit=444,file='plane_psi')
951         z = nz/2
952         do y=1,ny
953           do x=1,nx
954             write(444,*)x,y,psi(x,y,z)
955           end do
956           write(444,*)
957         end do
958         close(444)
959         return
960       end if
961     end do ! outer iteration
962 
963     ! We didnt converge in outer_it iterations
964     write(*,*)"crap... we didnt converge in ", outer_it, "iterations. :'-("
965     write(*,*)"k_err: ",k_err,"flux error: ",flux_err
966     write(*,*)"k=",keff
967 
968   end subroutine cmfd_solve
969 ! ==============================================================================
970   subroutine cmfd_lu
971     integer :: x,y,z,grp
972     real(dknd) :: m
973     ! perform an LU decomposition for each strip along the x direction.
974     do grp=1,g
975       do z=1,nz
976         do y=1,ny
977           x=1
978           lu_ud(x,y,z,grp) = diag(x,y,z,grp)
979           do x=2,nx
980             m = coup(2,x,y,z,grp)/lu_ud(x-1,y,z,grp)
981             !    west
982             lu_ll(x,y,z,grp) = m
983             !                     east
984             lu_ud(x,y,z,grp) = diag(x,y,z,grp)-m*coup(1,x-1,y,z,grp)
985           end do ! x
986           do x=1,nx-1
987             !          east
988             lu_uu(x,y,z,grp) = coup(1,x,y,z,grp)
989           end do ! x
990         end do ! y
991       end do ! z
992     end do ! group
993   end subroutine cmfd_lu
994 ! ==============================================================================
995   subroutine cmfd_bank
996     integer :: i,ix,iy,iz,ii,pos,nsrci,x,y,z,zeros
997     logical :: point_out,out_any
998     real(dknd) :: xx,yy,zz,psi_sum,h_cmfd,h_sample,log2
999     real(dknd),allocatable :: h_temp(:,:,:)
1000 
1001     allocate( h_temp(nx,ny,nz) )
1002 
1003     point_out = .false.
1004     out_any   = .false.
1005 
1006     log2 = log(two)
1007 
1008     fsrc_pop = 0
1009     sample_wgt = 0.0
```

```
1010      nsrci    = 0 ! number of source points inside the active mesh
1011
1012
1013      ! loop through all source points
1014      write(*,*)"origin: ",mesh_orig
1015
1016      ! open a file for plotting the fission source
1017      open(222,file='fso_old')
1018      open(223,file='fso_new')
1019
1020      do i=1,fso_src_count
1021        ! grab x,y,z position
1022        xx = fso_src(FSO_XXX,i)
1023        yy = fso_src(FSO_YYY,i)
1024        zz = fso_src(FSO_ZZZ,i)
1025        ! locate the source position in mesh
1026        ix = int( (xx-mesh_orig(1)) / hx ) + 1
1027        iy = int( (yy-mesh_orig(2)) / hy ) + 1
1028        iz = int( (zz-mesh_orig(3)) / hz ) + 1
1029        ii = ix + (iy-1)*nx + (iz-1)*nx*ny
1030        ! ensure that the point is inside the
1031        if (ix<1 .or. ix>nx) then
1032          point_out = .true.
1033          out_any   = .true.
1034        end if
1035        if (iy<1 .or. iy>ny) then
1036          point_out = .true.
1037          out_any   = .true.
1038        end if
1039        if (iz<1 .or. iz>nz) then
1040          point_out = .true.
1041          out_any   = .true.
1042        end if
1043        if (point_out .eq. .false.) then
1044          fsrc_pop(ii) = fsrc_pop(ii) + 1
1045          fsrc_pos(i) = ii
1046        else
1047          fsrc_pos(i) = -1
1048        end if
1049        point_out = .false.
1050      end do
1051      if (out_any .eq. .true.) then
1052        write(*,*)"Warning: there were fission source points outside of the mesh used for CMFD
              ."
1053      end if
1054
1055      ! plot fso_old
1056      iz = nz/2
1057      do iy=1,ny
1058        do ix=1,nx
1059          ii=nx*ny*(iz-1)+nx*(iy-1)+ix
1060          write(222,*)ix,iy,fsrc_pop(ii)
1061        end do
1062        write(222,*)
1063      end do
1064
1065      ! sweep through again to build wgt vector
1066      do i=1,fso_src_count
1067        pos = fsrc_pos(i)
1068        ii=pos
1069        if (pos .eq. -1) then
1070          sample_wgt(i) = 0
1071          cycle
1072        end if
1073        iz = int(pos/(nx*ny))
```

```
      pos = pos - iz*nx*ny
      iy = int(pos/nx)
      ix = pos - iy*nx
      if (fsrc_pop(ii)>0) then
        sample_wgt(i) = psi(ix,iy,iz)/fsrc_pop(ii)
      else
        sample_wgt(i) = psi(ix,iy,iz)
      end if
    end do

    ! now actually sample the nsrck points from the fission bank using the
    ! weights determined above
    call cmfd_sample(nsrck,fso_src_count,sample_wgt(1:fso_src_count),fsrc_ind)
    ! now do some Ministry of Truth work on the fso_src array
    fso_bnk = 0
    do i=1,nsrck
      fso_bnk(:,i) = fso_src(:,fsrc_ind(i))
    end do
    ! I think that's it. store fso_bnk back to fso_src, tell MCNP how many
    ! points are in there and call it good.
    fso_src_count = nsrck
    fso_src = fso_bnk

    ! determine the new source distribution and normalize
    call cmfd_pop()

    ! plot fso_new
    iz = nz/2
    do iy=1,ny
      do ix=1,nx
        ii=nx*ny*(iz-1)+nx*(iy-1)+ix
        write(223,*)ix,iy,fsrc_pop(ii)
      end do
      write(223,*)
    end do

    fsrc_pop = fsrc_pop/sum(fsrc_pop)

    ! calculate the source entropy of the actual and sampled fission source

    ! normalize psi
    psi = psi/sum(psi)

    where(psi /= 0)
      h_temp = psi*log(psi)
    else where
      h_temp = zero
    end where

    where(fsrc_pop /= 0)
      fsrc_pop = fsrc_pop*log(fsrc_pop)
    else where
      fsrc_pop = zero
    end where
    h_sample = -sum(fsrc_pop)/log2
    h_cmfd   = -sum(h_temp)/log2

    deallocate( h_temp )

    write(*,*)"FSD Entropy: ",h_cmfd
    write(*,*)"Sampled Entropy: ",h_sample

    ! check for regions with zero source points
    zeros = 0
    do i=1,n
```

```
      if(fsrc_pop(i)==0)then
        zeros = zeros+1
      end if
    end do
    if(zeros>0) then
      write(*,*)"There were",zeros," out of",n,"mesh regions with no source points sampled."
    end if

    return
  end subroutine cmfd_bank
! ======================================================================
  subroutine cmfd_sample( N, M, wgt, indx )
    !
    ! sample N items from M items with weights,
    ! save the indices of N selected items in array indx
    !
    ! range of N:  1...N
    ! range of M:  1...M
    ! The i-th of the M items has weight wgt(i), where the
    !   normalization of the weights is arbitrary
    ! indx(N):  N items, with indices in range 1..M
    !

    use mcnp_random, only: rang

    implicit none

    integer, intent(in) :: N     * number items needed
    integer, intent(in) :: M     * number items available
    real(dknd),intent(in) :: wgt(M)  * weights for available items
    integer, intent(out):: indx(N)   * indices of selected items

    real(dknd) :: prob, cum, wtot
    integer  :: i, knt, k

    wtot = sum( wgt )
    knt  = 0
    cum  = 0
    do i=1,M

      prob = wgt(i) * real(N-knt,dknd) / (wtot-cum)
      k = prob + rang()

      indx(knt+1:knt+k) = i
      knt = knt + k
      cum = cum + wgt(i)
    end do
    if( knt==N-1 ) then
      ! in case of roundoff, may have to replicate last item
      knt = knt + 1
      indx(knt) = M
    endif
    if( knt /= N ) then
      write(*,*) "***** count error in sample_N_from_M_weighted"
      stop
    endif
    return
  end subroutine cmfd_sample
! ======================================================================
  subroutine cmfd_pop()
    integer :: ix,iy,iz,ii,i
    real(dknd) :: xx,yy,zz
    logical :: point_out,out_any

    point_out = .false.
```

```
    out_any = .false.

    fsrc_pop = 0
    do i=1,fso_src_count
      ! grab x,y,z position
      xx = fso_src(FSO_XXX,i)
      yy = fso_src(FSO_YYY,i)
      zz = fso_src(FSO_ZZZ,i)
      ! locate the source position in mesh
      ix = int( (xx-mesh_orig(1)) / hx ) + 1
      iy = int( (yy-mesh_orig(2)) / hy ) + 1
      iz = int( (zz-mesh_orig(3)) / hz ) + 1
      ii = ix + (iy-1)*nx + (iz-1)*nx*ny
      ! ensure that the point is inside the
      if (ix<1 .or. ix>nx) then
        point_out = .true.
        out_any   = .true.
      end if
      if (iy<1 .or. iy>ny) then
        point_out = .true.
        out_any   = .true.
      end if
      if (iz<1 .or. iz>nz) then
        point_out = .true.
        out_any   = .true.
      end if
      if (point_out .eq. .false.) then
        fsrc_pop(ii) = fsrc_pop(ii) + 1
        fsrc_pos(i) = ii
      else
        fsrc_pos(i) = -1
      end if
      point_out = .false.
    end do
  end subroutine cmfd_pop
end module
```