Problem Set 4 Solutions

22.211 Reactor Physics I

Due: 2 April 2012

Bryan Herman

Pin-cell Code. The following tasks should be performed by the Monte Carlo pin-cell code:

1. Add input for LWR pellet surrounded by gap, clad, and coolant geometry 2.
2. Homogenize non-fuel regions (volume weighting) to obtain number densities of fuel & "moderator" 3.
3. Assume Dancoff factor=0.277, and use Carlvik's two-term rational approximation 4.
4. Get cross section data (PENDF files interpolator) for isotopes from Stellar site 5.
5. Start neutrons in fuel pellet from Chi spectrum and follow neutrons including:
   (a) Elastic down scatter in hydrogen, oxygen, zirconium (all Zr-90), U-235 and U-238 (remember mass dependence of scattering kernel)
   (b) Capture in hydrogen, U-235, and U-238 Thermal free gas scattering for hydrogen only (all other isotopes with asymptotic model)
   (c) Override PENDF U-238 absorption with data from your SLBW model (below ~ 1 keV)
   (d) For elastic scattering in U-235 and U-238, use potential scattering only
   (e) Assume temperatures of all isotopes are 300K
   (f) Follow neutrons to 1.e-5 eV and push back up to 1.1e-5 to assure no lost neutrons
6. Add tally of total fissions and variance (by history) to estimate kinf and variance
7. Add general tallies of two-group cross sections

The following dimensions and material properties should be used:

- Radius of the fuel, $r_f = 0.4096\,\text{cm}$
- Radius of inner clad, $r_{ci} = 0.4178\,\text{cm}$
- Radius of outer clad, $r_{co} = 0.4750\,\text{cm}$
- Density of fuel, $\rho_f = 10.2\,\text{g/cm}^3$
- Ignore Helium in gap
- Density of clad (all Zr-90), $\rho_c = 6.549\,\text{g/cm}^3$
- U-235 Enrichment, $\chi = 0.03035$

All questions were answered from a 10 million particle simulation.

Question 1. What is k-infinity and its variance (or std dev)?

$$k_{inf}\,(\text{analog}) = 1.38645 \pm 0.00038$$

$$k_{inf}\,(\text{collision}) = 1.38617 \pm 0.00043$$

Question 2. What is the cell-averaged fast-to-thermal flux ratio?

$$\frac{\phi_F}{\phi_T} = 5.13.$$

Question 3. What are the 2-group cell-averaged macroscopic cross sections?

Macro	Group 2 [cm^{-1}]	Group 1 [cm^{-1}]
Σ_t	1.4050	0.6451
Σ_s	1.2929	0.6353
Σ_a	0.1121	0.0098
Σ_f	0.0783	0.0026
$\nu\Sigma_f$	0.1923	0.0064
$\Sigma_s^{1\to 2}$	0.0219	

Question 4. What are the cell-average diffusion coefficients, computed by the integrating 3 different approximations from Lecture 10? (total, transport, 1/transport)

- $\langle D \rangle = \frac{1}{3\langle \Sigma_t \rangle}$

  $D_2 = 0.2373\,\text{cm} \qquad D_1 = 0.5167\,\text{cm}$

- $\langle D \rangle = \frac{1}{3\langle \Sigma_{tr} \rangle} = \frac{1}{3\langle \Sigma_t - \bar{\mu}\Sigma_s \rangle}$

  $D_2 = 0.4825\,\text{cm} \qquad D_1 = 0.9204\,\text{cm}$

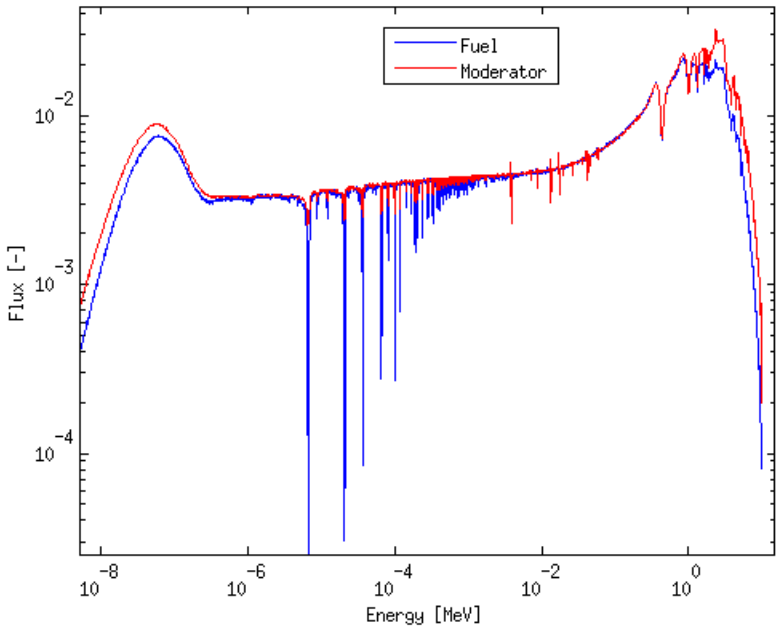
- $\langle D \rangle = \left\langle \frac{1}{3\Sigma_{tr}} \right\rangle = \left\langle \frac{1}{3(\Sigma_t - \bar{\mu}\Sigma_s)} \right\rangle$

  $D_2 = 0.5112\,\text{cm} \qquad D_1 = 1.3050\,\text{cm}$

Question 5. Tabulate 13-bin moderator/fuel flux ratio (remember to divide volumes).

Bin (eV)	Moderator/Fuel Ratio
0 - 0.1	1.28
0.1 - 0.5	1.09
0.5 - 1	1.03
1-6	1.02
6-10	1.19
10-25	1.07
25-50	1.07
50-100	1.05
100-1000	1.04
1000-10000	1.01
10000 - 100000	1.00
100000 - 500000	0.99
500000 - 10^7	1.27

Question 6. Plots (>1000 bins) of average fuel and moderator flux spectrum. He we have 5000 bins plotted.



A Input Files and Processing Codes

A.1 XML Input File

```
<?xml version="1.0"?>
<input>

<!-- Settings Information -->
  <settings>
    <histories> 10000000 </histories>
    <seed> 11 </seed>
    <source_type> 1 </source_type>
    <source_path> /home/bherman/Documents/Spring2012/211/SlowMC/lib/fission.h5 </source_pat
    <Dancoff> 0.277 </Dancoff>
    <res_iso> U-238 </res_iso>
    <radius> 0.4096 </radius>
  </settings>

  <materials>
    <material>
      <type> fuel </type>
      <V> 0.527072 </V>
      <nuclide name="U-238" N="0.0221" A="238" V="0.5271" thermal="false" >
        <path> /home/bherman/Documents/Spring2012/211/SlowMC/lib/U_238_300.h5 </path>
      </nuclide>
      <nuclide name="U-235" N="6.9915e-4" A="235" V="0.5271" thermal="false" >
        <path> /home/bherman/Documents/Spring2012/211/SlowMC/lib/U_235.h5 </path>
      </nuclide>
      <nuclide name="O-16" N="0.0455" A="16" V="0.5271" thermal="false" >
        <path> /home/bherman/Documents/Spring2012/211/SlowMC/lib/O_16.h5 </path>
      </nuclide>
    </material>
    <material>
      <type> moderator </type>
      <V> 1.06053 </V>
      <nuclide name="H-1" N="0.0666" A="1" V="0.878778" thermal="true" >
        <path> /home/bherman/Documents/Spring2012/211/SlowMC/lib/H_1.h5 </path>
      </nuclide>
      <nuclide name="O-16" N="0.0333" A="16" V="0.878778" thermal="false" >
        <path> /home/bherman/Documents/Spring2012/211/SlowMC/lib/O_16.h5 </path>
      </nuclide>
      <nuclide name="Zr-90" N="0.0439" A="90" V="0.160435" thermal="false" >
        <path> /home/bherman/Documents/Spring2012/211/SlowMC/lib/Zr_90.h5 </path>
      </nuclide>
    </material>
  </materials>

  <tallies>
    <tally type="flux" >
      <Ebins> 1e-11 0.625e-6 20.0 </Ebins>
    </tally>
    <tally type="total" >
```

```
      <Ebins> 1e-11 0.625e-6 20.0 </Ebins>
    </tally>
    <tally type="absorption" >
      <Ebins> 1e-11 0.625e-6 20.0 </Ebins>
    </tally>
    <tally type="scattering" >
      <Ebins> 1e-11 0.625e-6 20.0 </Ebins>
    </tally>
    <tally type="fission" >
      <Ebins> 1e-11 0.625e-6 20.0 </Ebins>
    </tally>
    <tally type="nufission" >
      <Ebins> 1e-11 0.625e-6 20.0 </Ebins>
    </tally>
    <tally type="diffusion" >
      <Ebins> 1e-11 0.625e-6 20.0 </Ebins>
    </tally>
    <tally type="transport">
      <Ebins> 1e-11 0.625e-6 20.0 </Ebins>
    </tally>
    <tally type="flux" dv="true">
      <Ebins> 1e-11 0.1e-6 0.5e-6 1e-6 6e-6 10e-6 25e-6 50e-6 100e-6 1e-3 1e-2 0.1 0.5 10 <
    </tally>
  </tallies>

</input>
```

A.2 Pre-processing Code

```
% Pre-processing script for material number densities

% Avagadros Number
Na = 0.6022;

% Mass Densities
rho_fuel = 10.2;  % UO2
rho_clad = 6.549; % Zr-90
rho_cool = 0.9966; % H2O

% Enrichment (w% U235/w% U)
enr = 0.03035;

% Molar Masses
M25 = 235.0439231;
M28 = 238.0507826;
M16 = 15.9949146;
M90 = 89.9047037;
MH1 = 1.0078250;

% Calculate Molar Mass of U
MU = (enr/M25 + (1-enr)/M28)^-1;
```

```
% Weight percent of Uranium
wU = MU/(MU + 2*M16);

% Number Density of Fuel
N25 = rho_fuel*(wU*enr)*Na/M25;
N28 = rho_fuel*(wU*(1-enr))*Na/M28;
N16 = rho_fuel*(1-wU)*Na/M16;

% Number Density of Coolant
Mw = 2*MH1 + M16;
NH = rho_cool*2*Na/Mw;
NO = rho_cool*Na/Mw;

% Number Density of Clad
NZr = rho_clad*Na/M90;
```

A.3 Post-processing Code

```
% Post Processing Script for SlowMC code

% get reaction rates
flux    = h5read('output.h5',horzcat('/tally_',num2str(1),'/mean'));
totRR   = h5read('output.h5',horzcat('/tally_',num2str(2),'/mean'));
absRR   = h5read('output.h5',horzcat('/tally_',num2str(3),'/mean'));
scatRR  = h5read('output.h5',horzcat('/tally_',num2str(4),'/mean'));
fissRR  = h5read('output.h5',horzcat('/tally_',num2str(5),'/mean'));
nfissRR = h5read('output.h5',horzcat('/tally_',num2str(6),'/mean'));
diffRR  = h5read('output.h5',horzcat('/tally_',num2str(7),'/mean'));
transRR = h5read('output.h5',horzcat('/tally_',num2str(8),'/mean'));

% fast to thermal
fast_to_thermal = sum(flux(2,:))/sum(flux(1,:));

% compute cross sections
totxs   = sum(totRR,2)./sum(flux,2);
absxs   = sum(absRR,2)./sum(flux,2);
scatxs  = sum(scatRR,2)./sum(flux,2);
fissxs  = sum(fissRR,2)./sum(flux,2);
nfissxs = sum(nfissRR,2)./sum(flux,2);
transxs = sum(transRR,2)./sum(flux,2);

% compute estimates of diffusion coefficient
diff_iso = 1./(3*totxs);
diff_trans = 1./(3*transxs);
diffcof = sum(diffRR,2)./sum(flux,2);

% compute kinf from xs
kinf = sum(sum(nfissRR)) / sum(sum(absRR));

% compute effective group 1->2 scattering xs
```

```
scat12xs = (nfissxs(2)/kinf - absxs(2))/(1 - nfissxs(1)/(kinf*absxs(1)));

% extract flux finer distribution
flux2 = h5read('output.h5',horzcat('/tally_',num2str(9),'/mean'));

% compute flux ratio mod/fuel
flux_rat = flux2(:,2)./flux2(:,1);

% extract energy and spectrum mean
E_spec = logspace(-11,log10(20.0),5000);
mean_spec = h5read('output.h5',horzcat('/tally_',num2str(10),'/mean'));

% plot spectrum
loglog(E_spec,mean_spec(:,1))
hold on
loglog(E_spec,mean_spec(:,2),'r')
```

B Main Codes

```
program main
!=============================================================================!
!> @mainpage SlowMC: Slowing Down Monte Carlo
!>
!> @section Overview
!>
!> This program solves the slowing down neutron transport equation in either
!> infinite medium or effective two-region collision probability theory. It
!> models parts of the same physics performed by the NJOY data processing code.
!> This code is for strictly academic purposes and allows the user to see the
!> relative impact of physics in the generation of multigroup cross sections
!> and on flux spectra. This code currently uses the following external
!> libraries:
!>  - HDF5 v1.8.#
!>
!> The package HDF5 can be downloaded from http://www.hdfgroup.org/HDF5/
!>
!> @section Compiling
!>
!> Compiling is as easy as running the Makefile with:
!>
!> @verbatim
!>   make xml-fortran
!>   make slowmc
!> @endverbatim
!>
!> @section Running
```

```
!>
!> To run SlowMC, execute the following:
!>
!> @verbatim
!>   slowmc
!> @endverbatim
!>
!=============================================================================!

  implicit none

  ! initialize problem
  call initialize()

  ! run problem
  call run_problem()

  ! finalize problem
  call finalize()

  ! terminate program
  stop

contains

!=============================================================================
! INTIALIZE
!> @brief high level routine for intializing problem
!=============================================================================

  subroutine initialize()

    use hdf5
    use global,    only: seed,allocate_problem,mat,tal,emax,emin,time_init,    &
   &                     compute_macro_cross_sections
    use input,     only: read_input
    use materials, only: compute_macroxs
    use output,    only: print_heading
    use timing,    only: timer_start,timer_stop

    ! local variables
    integer :: error ! hdf5 error
    real(8) :: rn    ! initial random number

    ! begin timer
    call timer_start(time_init)
```

```
    ! initialize the fortran hdf5 interface
    call h5open_f(error)

    ! print heading information
    call print_heading()

    ! read input
    call read_input()

    ! initalize random number generator
    rn = rand(seed)

    ! precompute macroscopic cross section of materials
    call compute_macro_cross_sections()

    ! end timer
    call timer_stop(time_init)

  end subroutine initialize

!===============================================================================
! RUN_PROBLEM
!> @brief main routine for executing the transport calculation
!===============================================================================

  subroutine run_problem()

    use global,    only: nhistories,mat,neut,eidx,emin,bank_tallies,time_run
    use particle,  only: init_particle
    use physics,   only: sample_source,perform_physics,get_eidx
    use timing,    only: timer_start,timer_stop

    ! local variables
    integer :: i  ! iteration counter

    ! begin timer
    call timer_start(time_run)

    ! begin loop over histories
    do i = 1,nhistories

      ! intialize history
      call init_particle(neut)

      ! sample source energy
      call sample_source()
```

```
      ! begin transport of neutron
      do while (neut%alive)

        ! check for energy cutoff
        if (neut%E < emin) neut%E = 1.1e-11_8

        ! call index routine for first tally
        eidx = get_eidx(neut%E)

        ! perform physics and also records collision tally
        call perform_physics()

      end do

      ! neutron is dead if out of transport loop (ecut or absorb) --> bank tally
      call bank_tallies()

      ! print update to user
      if (mod(i,nhistories/10) == 0) then
        write(*,'(/A,1X,I0,1X,A)') 'Simulated',i,'neutrons...'
      end if

    end do

    ! end timer
    call timer_stop(time_run)


  end subroutine run_problem

!===============================================================================
! FINALIZE
!> @brief routine that finalizes the problem
!===============================================================================

  subroutine finalize()

    use global, only: finalize_tallies,deallocate_problem
    use hdf5
    use output, only: write_output

    ! local variables
    integer :: error ! hdf5 error

    ! calculate statistics on tallies
    call finalize_tallies()
```

```
    ! write output
    call write_output()

    ! deallocate problem
    call deallocate_problem()

    ! close the fortran interface
    call h5close_f(error)

  end subroutine finalize

end program main
```

C Module Files

C.1 Global

```
!===============================================================================!
! MODULE: global
!
!> @author Bryan Herman
!>
!> @brief Contains all of the global variables
!===============================================================================!

module global

  use materials, only: material_type
  use particle,  only: particle_type
  use tally,     only: tally_type
  use timing,    only: Timer

  implicit none
  save

  ! version information
  integer :: VERSION_MAJOR   = 0
  integer :: VERSION_MINOR   = 1
  integer :: VERSION_RELEASE = 1

  ! list all types
  type(particle_type)              :: neut
  type(material_type), allocatable :: mat(:)
```

```
  type(tally_type), allocatable    :: tal(:)

  ! list history input information
  integer :: nhistories
  integer :: seed
  integer :: source_type

  ! list global vars that are set during run
  integer :: eidx         ! energy index for cross sections
  integer :: n_tallies    ! number of tallies
  integer :: n_materials ! n_materials
  integer :: res_iso      ! resonant isotope id in material 1
  real(8) :: Dancoff      ! lattice Dancoff factor (C)
  real(8) :: radius       ! radius of fuel pin

  ! set max and min energy
  real(8) :: emin = 1e-11_8
  real(8) :: emax = 20.0_8

  ! kT value base on 300K
  real(8) :: kT = 8.6173324e-5_8*300*1.0e-6_8

  ! set nu value
  real(8) :: nubar = 2.455_8

  ! timers
  type(Timer) :: time_init
  type(Timer) :: time_run

  ! analog counters for k-inf
  integer :: n_abs=0.0_8
  integer :: n_fiss=0.0_8
  real(8) :: ana_kinf_mean = 0.0_8
  real(8) :: ana_kinf_std  = 0.0_8
  real(8) :: col_kinf_mean = 0.0_8
  real(8) :: col_kinf_std  = 0.0_8

contains

!===============================================================================
! ALLOCATE_PROBLEM
!> @brief allocates global variables for calculation
!===============================================================================

  subroutine allocate_problem()

    ! formal variables
```

```
    ! allocate tallies
    if (.not.allocated(tal)) allocate(tal(n_tallies))
    if (.not.allocated(mat)) allocate(mat(n_materials))

  end subroutine allocate_problem

!===============================================================================
! DEALLOCATE_PROBLEM
!> @brief deallocates global variables
!===============================================================================

  subroutine deallocate_problem()

    use materials, only: deallocate_material
    use tally,     only: deallocate_tally

    ! local variables
    integer :: i  ! loop counter

    ! deallocate within materials
    do i = 1,n_materials

      ! deallocate material
      call deallocate_material(mat(i))

    end do

    ! deallocate material variable
    if (allocated(mat)) deallocate(mat)

    ! deallocate within tallies
    do i = 1,n_tallies

      ! deallocate tally
      call deallocate_tally(tal(i))

    end do

    ! deallocate tally variable
    if (allocated(tal)) deallocate(tal)

  end subroutine deallocate_problem

!===============================================================================
! COMPUTE_MACRO_CROSS_SECTIONS
!> @brief routine that handles the call to compute macro cross sections
```

```
!===============================================================================

  subroutine compute_macro_cross_sections()

    use materials, only: compute_macroxs

    ! local variables
    integer :: i  ! loop counter

    ! begin loop over materals
    do i = 1,n_materials

      ! call routine to compute xs
      call compute_macroxs(mat(i))

    end do

  end subroutine compute_macro_cross_sections

!===============================================================================
! ADD_TO_TALLIES
!> @brief routine that adds temporary value to tallies
!===============================================================================

  subroutine add_to_tallies()

    use tally, only: add_to_tally

    ! local variables
    integer :: i             ! loop counter
    real(8) :: fact = 1.0_8 ! multiplier factor
    real(8) :: totxs         ! total macroscopic xs of material
    real(8) :: mubar         ! average cosine scattering angle

    ! compute macroscopic cross section
    totxs = sum(mat(neut%region)%totalxs(eidx,:))

    ! begin loop over tallies
    do i = 1,n_tallies

      ! set multiplier
      select case(tal(i)%react_type)

        ! flux only
        case(0)
          fact = 1.0_8
```

```
      ! total
      case(1)
        fact = totxs

      ! absorption
      case(2)
        fact = sum(mat(neut%region)%absorxs(eidx,:))

      ! scattering
      case(3)
        fact = sum(mat(neut%region)%scattxs(eidx,:))

      ! nufission
      case(4)
        fact = nubar*sum(mat(neut%region)%fissixs(eidx,:))

      ! fission
      case(5)
        fact = sum(mat(neut%region)%fissixs(eidx,:))

      ! diffusion coefficient
      case(6)
        fact = 1._8/(3._8*sum(mat(neut%region)%transxs(eidx,:)))

      ! transport
      case(7)
        fact = sum(mat(neut%region)%transxs(eidx,:))

      ! micro capture
      case(8)
        if (neut%region == tal(i)%region) then
          fact = mat(neut%region)%isotopes(tal(i)%isotope)%xs_capt(eidx)
        else
          cycle
        end if

      ! default is flux tally
      case DEFAULT
        fact = 1.0_8

    end select

    ! call routine to add tally
    call add_to_tally(tal(i),fact,totxs,neut%E,neut%region)

  end do
```

```
  end subroutine add_to_tallies

!===============================================================================
! BANK_TALLIES
!> @brief routine that record temporary history information in tallies
!===============================================================================

  subroutine bank_tallies()

    use tally, only: bank_tally

    ! local variables
    integer :: i  ! loop counter

    ! begin loop over tallies
    do i = 1,n_tallies

      ! call routine to bank tally
      call bank_tally(tal(i))

    end do

  end subroutine bank_tallies

!===============================================================================
! FINALIZE_TALLIES
!> @brief routine that calls another routine to compute tally statistics
!===============================================================================

  subroutine finalize_tallies()

    use tally, only: calculate_statistics

    ! local variables
    integer :: i ! loop counter
    integer :: j ! loop counter

    ! begin loop over tallies
    do i = 1,n_tallies

      ! call routine to compute statistics
      call calculate_statistics(tal(i),nhistories)

      ! normalize by volumes and histories if flux tally
      if (tal(i)%flux_tally .or. tal(i)%dv) then
        do j = 1,n_materials
          tal(i)%mean(:,j) = tal(i)%mean(:,j) / mat(j)%vol
```

```
        end do
      end if

    end do

    ! compute k_inf
    ana_kinf_mean = dble(n_fiss)*nubar/dble(nhistories)
    ana_kinf_std  = nubar*sqrt((dble(n_fiss)/dble(nhistories) -                &
   &                (dble(n_fiss)/dble(nhistories))**2)/dble(nhistories-1))

!   ana_kinf_std  = 0.0_8
    col_kinf_mean = sum(tal(n_tallies)%mean)
    col_kinf_std  = sum(tal(n_tallies)%std)

  end subroutine finalize_tallies

end module global
```

C.2 Input

```
!==============================================================================!
! MODULE: input
!
!> @author Bryan Herman
!>
!> @brief Handles reading in the input xml file and intializing global vars
!==============================================================================!

module input

  implicit none
  private
  public read_input

contains

!===============================================================================
! READ_INPUT
!> @brief Reads the input xml file and sets global variables
!===============================================================================

  subroutine read_input

    use global,           only: nhistories,seed,source_type,mat,emin,emax,     &
   &                            allocate_problem,tal,n_tallies,n_materials,    &
```

```
&                              res_iso,Dancoff,radius
 use materials,          only: setup_material,load_source,load_isotope
 use tally,              only: set_user_tally,set_spectrum_tally,              &
&                              set_kinf_tally
 use xml_data_input_t

 ! local variables
 logical                        :: file_exists  ! see if file exists
 character(len=255)             :: filename     ! filename to open
 real(8)                        :: N            ! temp number dens
 real(8)                        :: A            ! temp atomic weight
 real(8)                        :: vol          ! volume of region
 character(len=255)             :: path         ! path to isotope file
 character(len=255)             :: name         ! name of isotope
 logical                        :: thermal      ! contains thermal lib
 integer                        :: i            ! iteration counter
 integer                        :: j            ! iteration counter
 integer                        :: nisotopes    ! number of isotopes in mat
 integer                        :: react_type   ! reaction type
 integer                        :: isotope=0    ! isotope for micro mult
 integer                        :: region=0     ! region to tally micros
 real(8), allocatable           :: Ebins(:)     ! tally energy bins
 logical                        :: dv           ! divide tally by volumes

 ! check for input file
 filename = "input.xml"
 inquire(FILE=trim(filename), EXIST=file_exists)
 if (.not. file_exists) then
   write(*,*) 'Cannot read input file!'
   stop
 else

   ! tell user
   write(*,'(A/)') "Reading INPUT XML file..."

 end if

 ! read in input file
 call read_xml_file_input_t(trim(filename))

 ! read in settings
 nhistories = settings_%histories
 seed = settings_%seed
 source_type = settings_%source_type

 ! get size of materials
 n_materials = size(materials_%material)
```

```
! get size of tallies
if (.not.associated(tallies_%tally)) then
  n_tallies = 2
else
  n_tallies = size(tallies_%tally) + 2
end if

! allocate problem
call allocate_problem()

! begin loop around materials
do i = 1,n_materials

  ! get number of isotopes and volume
  nisotopes = size(materials_%material(i)%nuclides)
  vol = materials_%material(i)%V

  ! set homogeneous volume
  if (trim(materials_%material(i)%type)=='homogeneous') vol = 1.0_8

  ! set up the material object
  call setup_material(mat(i),emin,emax,nisotopes,vol)

  ! begin loop over isotope materials
  do j = 1,mat(i)%nisotopes

    ! check volumes and number densities
    if (trim(materials_%material(i)%type)=='homogeneous') then

      ! set volume to 1 and don't adjust n dens
      N = materials_%material(i)%nuclides(j)%N

    else if (trim(materials_%material(i)%type)=='fuel') then

      ! don't adjust n dens
      N = materials_%material(i)%nuclides(j)%N

      ! check volume
      if (abs(vol - 0.0_8) < 1e-10_8) then
        write(*,*) 'Please enter a physical fuel volume!'
        stop
      end if

    else

      ! check volume
```

```
      if (abs(vol - 0.0_8) < 1e-10_8) then
        write(*,*) 'Please enter a physical moderator volume!'
        stop
      end if

      ! adjust number density by volume weighting
      N = materials_%material(i)%nuclides(j)%N*                    &
     &    (materials_%material(i)%nuclides(j)%V/vol)

    end if

    ! extract other info
    A = materials_%material(i)%nuclides(j)%A
    path = materials_%material(i)%nuclides(j)%path
    thermal = materials_%material(i)%nuclides(j)%thermal
    name = materials_%material(i)%nuclides(j)%name

    ! load the isotope into memory
    call load_isotope(mat(i),N,A,path,thermal,name)

    ! check for resonant isotope in material 1
    if (trim(materials_%material(i)%type)=='fuel' .and.            &
   &    trim(settings_%res_iso) == trim(name)) then

      ! get Dancoff factor and resonant isotope
      res_iso = j
      Dancoff = settings_%Dancoff
      radius = settings_%radius

    end if

  end do

end do

! begin loop over tallies
do i = 1,n_tallies-2

  ! check for divide by volume
  dv = .false.
  if (tallies_%tally(i)%dv) dv = .true.

  ! set reaction type
  select case(trim(tallies_%tally(i)%type))
    case('flux')
      react_type = 0
    case('total')
```

```
        react_type = 1
      case('absorption')
        react_type = 2
      case('scattering')
        react_type = 3
      case('nufission')
        react_type = 4
      case('fission')
        react_type = 5
      case('diffusion')
        react_type = 6
      case('transport')
        react_type = 7
      case('micro_capture')
        react_type = 8
        isotope = tallies_%tally(i)%isotope
        region = tallies_%tally(i)%region
      case DEFAULT
        react_type = 0
    end select

    ! preallocate Ebins
    if(.not. allocated(Ebins)) allocate(Ebins(size(tallies_%tally(i)%Ebins)))

    ! set Ebins
    Ebins = tallies_%tally(i)%Ebins

    ! set up user-defined tallies
    call set_user_tally(tal(i),Ebins,size(Ebins),react_type,isotope,region,  &
   &                    n_materials,dv)

    ! deallocate Ebins
    if(allocated(Ebins)) deallocate(Ebins)

  end do

  ! set up spectrum tally
  call set_spectrum_tally(tal(n_tallies-1),emax,emin,n_materials)

  ! set up k_inf tally
  call set_kinf_tally(tal(n_tallies),emax,emin,n_materials)

  ! load the source
  call load_source(mat(1),source_type,settings_%source_path)

end subroutine read_input
```

```
end module input
```

C.3 Materials

```
!=============================================================================!
! MODULE: materials
!
!> @author Bryan Herman
!>
!> @brief Contains information about the isotopics of problem
!=============================================================================!

module materials

  implicit none
  private
  public :: setup_material,load_source,load_isotope,compute_macroxs,          &
 &          deallocate_material

  type :: source_type

    real(8), allocatable :: E(:)      ! energy range for fission source
    real(8)              :: cdf_width ! width of cdf bins from 0 to 1

  end type source_type

  type :: thermal_type

    integer              :: kTsize    ! size of kT vector
    integer              :: cdfsize   ! size of cdf
    real(8), allocatable :: kTvec(:)  ! vector of kT values
    real(8), allocatable :: Erat(:,:) ! energy
    real(8)              :: cdf_width ! width of cdf interval from 0 to 1

  end type thermal_type

  type :: iso_type

    real(8)              :: N          ! number density
    real(8)              :: A          ! atomic weight
    real(8)              :: alpha      ! (A-1)^2/(A+1)^2
    real(8)              :: mubar      ! average cosine scattering angle
    real(8), allocatable :: xs_capt(:) ! capture micro xs
    real(8), allocatable :: xs_scat(:) ! scattering micro xs
    real(8), allocatable :: xs_fiss(:) ! fission micro xs
```

```
    character(len=255)     :: name         ! name of isotope

    logical                :: thermal      ! thermal scatterer
    type(thermal_type)     :: thermal_lib  ! thermal library

  end type iso_type

  type, public ::  material_type

    type(source_type)           :: source         ! the source of neutrons
    type(iso_type), allocatable :: isotopes(:)    ! 1-D array of isotopes in mat
    integer                     :: nisotopes      ! number of isotopes in mat
    integer                     :: curr_iso       ! the current isotope
    integer                     :: npts           ! number of points in energy
    real(8)                     :: E_width        ! width of energy interval
    real(8)                     :: E_min          ! min energy
    real(8)                     :: E_max          ! max energy
    real(8)                     :: vol            ! volume of region
    real(8), allocatable        :: totalxs(:,:)   ! array of macroscopic tot xs
    real(8), allocatable        :: scattxs(:,:)   ! array of macroscopic scat xs
    real(8), allocatable        :: absorxs(:,:)   ! array of macroscopic abs xs
    real(8), allocatable        :: captuxs(:,:)   ! array of macroscopic capt xs
    real(8), allocatable        :: fissixs(:,:)   ! array of macroscopic fiss xs
    real(8), allocatable        :: transxs(:,:)   ! array of macro transport xs

  end type material_type

contains

!===============================================================================
! SET_UP_MATERIALS
!> @brief routine that initializes the materials
!===============================================================================

  subroutine setup_material(this,emin,emax,nisotopes,vol)

    ! formal variables
    type(material_type) :: this      ! a material
    real(8)             :: emin      ! minimum energy to consider
    real(8)             :: emax      ! maximum energy to consider
    real(8)             :: vol       ! volume of material
    integer             :: nisotopes ! number of isotopes

    ! set number of isotopes
    this%nisotopes = nisotopes

    ! set volume
```

```
    this%vol = vol

    ! allocate isotopes array
    if (.not. allocated(this%isotopes)) allocate(this%isotopes(this%nisotopes))

    ! set up current isotope index
    this%curr_iso = 1

    ! set energy bounds
    this%E_min = emin
    this%E_max = emax

  end subroutine setup_material

!=============================================================================
! LOAD_ISOTOPE
!> @brief routine that loads isotope properties, xs, etc. into memory
!=============================================================================

  subroutine load_isotope(this,N,A,path,thermal,name)

    use hdf5

    ! formal variables
    type(material_type),target :: this    ! a material
    real(8)                    :: N       ! number density
    real(8)                    :: A       ! atomic weight
    character(len=255)         :: path    ! path to isotope
    character(len=255)         :: name    ! name of isotope
    logical                    :: thermal ! contains a thermal lib

    ! local variables
    integer                          :: error        ! hdf5 error
    integer(HID_T)                   :: hdf5_file    ! hdf5 file id
    integer(HID_T)                   :: dataset_id   ! hdf5 dataset id
    integer(HSIZE_T), dimension(1)   :: dim1         ! dimension of hdf5 var
    integer(HSIZE_T), dimension(2)   :: dim2         ! dimension of hdf5 var
    integer                          :: vecsize      ! vector size
    type(thermal_type), pointer      :: therm

    ! display to user
    write(*,*) 'Loading isotope: ',trim(name)

    ! set parameters
    this%isotopes(this%curr_iso)%N = N
    this%isotopes(this%curr_iso)%A = A
    this%isotopes(this%curr_iso)%mubar = 2._8/(3._8*A)
```

```
  this%isotopes(this%curr_iso)%alpha = ((A-1._8)/(A+1._8))**2
  this%isotopes(this%curr_iso)%thermal = thermal
  this%isotopes(this%curr_iso)%name = name

  ! open up hdf5 file
  call h5fopen_f(trim(path),H5F_ACC_RDWR_F,hdf5_file,error)

  ! read size of vector
  call h5dopen_f(hdf5_file,"/vecsize",dataset_id,error)
  dim1 = (/1/)
  call h5dread_f(dataset_id,H5T_NATIVE_INTEGER,vecsize,dim1,error)
  call h5dclose_f(dataset_id,error)

  ! allocate all xs vectors
  if (.not.allocated(this%isotopes(this%curr_iso)%xs_scat))                    &
&         allocate(this%isotopes(this%curr_iso)%xs_scat(vecsize))
  if (.not.allocated(this%isotopes(this%curr_iso)%xs_capt))                    &
&         allocate(this%isotopes(this%curr_iso)%xs_capt(vecsize))
  if (.not.allocated(this%isotopes(this%curr_iso)%xs_fiss))                    &
&         allocate(this%isotopes(this%curr_iso)%xs_fiss(vecsize))


  ! keep the size
  this%npts = vecsize

  ! zero out xs vectors
  this%isotopes(this%curr_iso)%xs_scat = 0.0_8
  this%isotopes(this%curr_iso)%xs_capt = 0.0_8
  this%isotopes(this%curr_iso)%xs_fiss = 0.0_8

  ! read in xs
  call h5dopen_f(hdf5_file,"/xs_scat",dataset_id,error)
  dim1 = (/vecsize/)
  call h5dread_f(dataset_id,H5T_NATIVE_DOUBLE,                                 &
&     this%isotopes(this%curr_iso)%xs_scat,dim1,error)
  call h5dclose_f(dataset_id,error)
  call h5dopen_f(hdf5_file,"/xs_capt",dataset_id,error)
  dim1 = (/vecsize/)
  call h5dread_f(dataset_id,H5T_NATIVE_DOUBLE,                                 &
&     this%isotopes(this%curr_iso)%xs_capt,dim1,error)
  call h5dclose_f(dataset_id,error)
  call h5dopen_f(hdf5_file,"/xs_fiss",dataset_id,error)
  dim1 = (/vecsize/)
  call h5dread_f(dataset_id,H5T_NATIVE_DOUBLE,                                 &
&     this%isotopes(this%curr_iso)%xs_fiss,dim1,error)
  call h5dclose_f(dataset_id,error)
```

```
! get energy interval width
call h5dopen_f(hdf5_file,"/E_width",dataset_id,error)
dim1 = (/1/)
call h5dread_f(dataset_id,H5T_NATIVE_DOUBLE,this%E_width,dim1,error)
call h5dclose_f(dataset_id,error)

! check for thermal scattering kernel and load that
if (this%isotopes(this%curr_iso)%thermal) then

  ! set pointer
  therm => this%isotopes(this%curr_iso)%thermal_lib

  ! load sizes
  call h5dopen_f(hdf5_file,"/kTsize",dataset_id,error)
  dim1 = (/1/)
  call h5dread_f(dataset_id,H5T_NATIVE_INTEGER,therm%kTsize,dim1,error)
  call h5dclose_f(dataset_id,error)
  call h5dopen_f(hdf5_file,"/cdfsize",dataset_id,error)
  dim1 = (/1/)
  call h5dread_f(dataset_id,H5T_NATIVE_INTEGER,therm%cdfsize,dim1,error)
  call h5dclose_f(dataset_id,error)

  ! read in cdf width
  call h5dopen_f(hdf5_file,"/cdf_width",dataset_id,error)
  dim1 = (/1/)
  call h5dread_f(dataset_id,H5T_NATIVE_DOUBLE,therm%cdf_width,dim1,error)
  call h5dclose_f(dataset_id,error)

  ! preallocate vectors
  if(.not.allocated(therm%kTvec)) allocate(therm%kTvec(therm%kTsize))
  if(.not.allocated(therm%Erat)) allocate(therm%Erat(therm%cdfsize,therm%↩
     kTsize))

  ! read in vectors
  call h5dopen_f(hdf5_file,"/kT",dataset_id,error)
  dim1 = (/therm%kTsize/)
  call h5dread_f(dataset_id,H5T_NATIVE_DOUBLE,therm%kTvec,dim1,error)
  call h5dclose_f(dataset_id,error)
  call h5dopen_f(hdf5_file,"/Erat",dataset_id,error)
  dim2 = (/therm%cdfsize,therm%kTsize/)
  call h5dread_f(dataset_id,H5T_NATIVE_DOUBLE,therm%Erat,dim2,error)
  call h5dclose_f(dataset_id,error)

end if

! close hdf5 file
call h5fclose_f(hdf5_file,error)
```

```
    ! increment isotope counter
    this%curr_iso = this%curr_iso + 1

  end subroutine load_isotope

!=======================================================================
! LOAD_SOURCE
!> @brief routine to load fission source into memory
!=======================================================================

  subroutine load_source(this,source_type,source_path)

    use hdf5

    ! formal variables
    type(material_type) :: this        ! a material
    integer             :: source_type ! 0 - fixed, 1 - fission
    character(len=255)  :: source_path ! path to source file

    ! local variables
    integer                         :: error       ! hdf5 error
    integer(HID_T)                  :: hdf5_file   ! hdf5 file id
    integer(HID_T)                  :: dataset_id  ! hdf5 dataset id
    integer(HSIZE_T), dimension(1) :: dim1        ! dimension of hdf5 var
    integer                         :: vecsize     ! vector size for fission


    ! check for fission source
    if (source_type == 1) then

      ! open the fission source file
      call h5fopen_f(trim(source_path),H5F_ACC_RDWR_F,hdf5_file,error)

      ! open dataset and read in vector size
      call h5dopen_f(hdf5_file,"/vecsize",dataset_id,error)
      dim1 = (/1/)
      call h5dread_f(dataset_id,H5T_NATIVE_INTEGER,vecsize,dim1,error)
      call h5dclose_f(dataset_id,error)

      ! open dataset and read in width of cdf interval
      call h5dopen_f(hdf5_file,"/cdf_width",dataset_id,error)
      dim1 = (/1/)
      call h5dread_f(dataset_id,H5T_NATIVE_DOUBLE,this%source%cdf_width,dim1,  &
     &               error)
      call h5dclose_f(dataset_id,error)
```

```
      ! preallocate vectors in source object
      if(.not.allocated(this%source%E)) allocate(this%source%E(vecsize))

      ! open dataset and read in energy vector
      call h5dopen_f(hdf5_file,"/E",dataset_id,error)
      dim1 = (/vecsize/)
      call h5dread_f(dataset_id,H5T_NATIVE_DOUBLE,this%source%E,dim1,error)
      call h5dclose_f(dataset_id,error)

      ! close the file
      call h5fclose_f(hdf5_file,error)

    end if

  end subroutine load_source

!=======================================================================
! COMPUTE_MACROXS
!> @brief routine to pre-compute macroscopic cross sections
!=======================================================================

  subroutine compute_macroxs(this)

    ! formal variables
    type(material_type),target :: this ! a material

    ! local variables
    integer                  :: i   ! loop counter
    type(iso_type), pointer :: iso ! pointer to current isotope

    ! allocate xs arrays
    if (.not.allocated(this%totalxs))                                    &
   &                          allocate(this%totalxs(this%npts,this%nisotopes))
    if (.not.allocated(this%scattxs))                                    &
   &                          allocate(this%scattxs(this%npts,this%nisotopes))
    if (.not.allocated(this%absorxs))                                    &
   &                          allocate(this%absorxs(this%npts,this%nisotopes))
    if (.not.allocated(this%captuxs))                                    &
   &                          allocate(this%captuxs(this%npts,this%nisotopes))
    if (.not.allocated(this%fissixs))                                    &
   &                          allocate(this%fissixs(this%npts,this%nisotopes))
    if (.not.allocated(this%transxs))                                    &
   &                          allocate(this%transxs(this%npts,this%nisotopes))

    ! zero out total xs
    this%totalxs = 0.0_8
```

```
    ! begin loop over isotopes
    do i = 1,this%nisotopes

      ! set pointer to isotope
      iso => this%isotopes(i)

      ! multiply microscopic cross section by number density and append
      this%captuxs(:,i) = iso%N*(iso%xs_capt)
      this%fissixs(:,i) = iso%N*(iso%xs_fiss)
      this%scattxs(:,i) = iso%N*(iso%xs_scat)
      this%absorxs(:,i) = iso%N*(iso%xs_capt + iso%xs_fiss)
      this%totalxs(:,i) = iso%N*(iso%xs_capt + iso%xs_fiss + iso%xs_scat)
      this%transxs(:,i) = this%totalxs(:,i) - iso%mubar*this%scattxs(:,i)

    end do

  end subroutine compute_macroxs

!===============================================================================
! DEALLOCATE_MATERIAL
!> @brief routine to deallocate a material
!===============================================================================

  subroutine deallocate_material(this)

    ! formal variables
    type(material_type) :: this ! a material

    ! local variables
    integer :: i ! loop counter

    ! deallocate source information
    if (allocated(this%source%E)) deallocate(this%source%E)

    ! begin loop over isotopes for deallocation
    do i = 1,this%nisotopes

      ! deallocate thermal library
      if (allocated(this%isotopes(i)%thermal_lib%kTvec)) deallocate           &
     &            (this%isotopes(i)%thermal_lib%kTvec)
      if (allocated(this%isotopes(i)%thermal_lib%Erat)) deallocate            &
     &            (this%isotopes(i)%thermal_lib%Erat)

      ! deallocate xs
      if (allocated(this%isotopes(i)%xs_scat)) deallocate                     &
     &            (this%isotopes(i)%xs_scat)
      if (allocated(this%isotopes(i)%xs_capt)) deallocate                     &
```

```
     &              (this%isotopes(i)%xs_capt)
      if (allocated(this%isotopes(i)%xs_fiss)) deallocate                    &
     &              (this%isotopes(i)%xs_fiss)
    end do

    ! deallocate isotopes
    if (allocated(this%isotopes)) deallocate(this%isotopes)

    ! deallocate macro xs
    if (allocated(this%totalxs)) deallocate(this%totalxs)
    if (allocated(this%scattxs)) deallocate(this%scattxs)
    if (allocated(this%absorxs)) deallocate(this%absorxs)
    if (allocated(this%captuxs)) deallocate(this%captuxs)
    if (allocated(this%fissixs)) deallocate(this%fissixs)
    if (allocated(this%transxs)) deallocate(this%transxs)

  end subroutine deallocate_material

end module materials
```

C.4 Output

```
!=============================================================================!
! MODULE: output
!
!> @author Bryan Herman
!>
!> @brief Contains routines for outputttting major info to user
!=============================================================================!

module output

  implicit none
  private
  public :: print_heading,write_output

contains

!=============================================================================
! PRINT_HEADING
!> @brief prints the code heading and run information
!=============================================================================

  subroutine print_heading()
```

```fortran
    use global, only: VERSION_MAJOR,VERSION_MINOR,VERSION_RELEASE

    ! local variables
    character(len=10) :: today_date
    character(len=8)  :: today_time

    ! write header
    write(*, FMT='(/9(A/))') &
  & ' .d8888b.  888                        888b     d888  .d8888b.     ',        &
  & 'd88P  Y88b 888                        8888b   d8888 d88P  Y88b    ',        &
  & 'd88P  Y88b 888                        8888b   d8888 d88P  Y88b    ',        &
  & 'd88P  Y88b 888                        8888b   d8888 d88P  Y88b    ',        &
  & ' "Y888b.   888  .d88b.  888  888  888 888Y88888P888 888         ',        &
  & '    "Y88b. 888 d88""88b 888  888  888 888 Y888P 888 888         ',        &
  & '      "888 888 888  888 888  888  888 888  Y8P  888 888    888    ',        &
  & 'Y88b  d88P 888 Y88..88P Y88b 888 d88P 888   "   888 Y88b  d88P   ',        &
  & ' "Y8888P"  888  "Y88P"   "Y8888888P"  888       888  "Y8888P"     '

    ! Write version information
    write(*, FMT=*)                                                              &
 &   '     Developed At:  Massachusetts Institute of Technology'
    write(*, FMT='(6X,"Version:",7X,I1,".",I1,".",I1)')                          &
 &        VERSION_MAJOR, VERSION_MINOR, VERSION_RELEASE

    ! Write the date and time
    call get_today(today_date, today_time)
    write(*, FMT='(6X,"Date/Time:",5X,A,1X,A)')                                  &
 &      trim(today_date), trim(today_time)


    ! write out divider
    write(*,FMT='(A/)') '-------------------------------------------------------'

  end subroutine print_heading

!===============================================================================
! WRITE_OUTPUT
!> @brief routine that writes timing info and hdf5 file
!===============================================================================

  subroutine write_output()

    use global, only: time_init,time_run,tal,n_tallies,ana_kinf_mean,          &
   &                  ana_kinf_std,col_kinf_mean,col_kinf_std
    use hdf5

    ! local variables
```

```
    integer                          :: i            ! loop counter
    integer                          :: error        ! hdf5 error
    integer(HID_T)                   :: hdfile       ! hdf5 file
    integer(HID_T)                   :: dataspace_id ! dataspace identifier
    integer(HID_T)                   :: dataset_id   ! dataset identifier
    integer(HID_T)                   :: group_id     ! group id
    integer(HSIZE_T), dimension(1)   :: dim1         ! vector for hdf5 dims
    integer(HSIZE_T), dimension(2)   :: dim2         ! matrix for hdf5 dims
    character(11)                    :: talnum       ! tally number

    ! write results header
    write(*,'(/A,/,A,/)') "Results","-----------------------"

    ! write timing information
    write(*,100) "Initialization time:",time_init%elapsed
    write(*,100) "Transport time:",time_run%elapsed
    write(*,*)

    ! format for time write statements
100 format (1X,A,T25,ES10.4," seconds")

    ! write k-inf information
    write(*,101) 'k-inf (analog):',ana_kinf_mean,ana_kinf_std
    write(*,101) 'k-inf (coll):',col_kinf_mean,col_kinf_std
    write(*,*)

    ! format for kinf write statements
101 format(1X,A,T25,F7.5,1X,'+/-',1X,F7.5)

    ! open up output hdf5 file
    call h5fcreate_f("output.h5",H5F_ACC_TRUNC_F,hdfile,error)

    ! begin loop around tallies to write out
    do i = 1,n_tallies

      ! get tally number
      write (talnum, '(I11)') i
      talnum = adjustl(talnum)

      ! open up a group
      call h5gcreate_f(hdfile,"tally_"//trim(talnum),group_id,error)

      ! write mean
      dim2 = (/size(tal(i)%mean,1),size(tal(i)%mean,2)/)
      call h5screate_simple_f(2,dim2,dataspace_id,error)
      call h5dcreate_f(hdfile,"tally_"//trim(talnum)//"/mean",H5T_NATIVE_DOUBLE&
     &                 ,dataspace_id,dataset_id,error)
```

```
      call h5dwrite_f(dataset_id,H5T_NATIVE_DOUBLE,tal(i)%mean,dim2,error)
      call h5sclose_f(dataspace_id,error)
      call h5dclose_f(dataset_id,error)

      ! write standard deviation
      dim2 = (/size(tal(i)%std,1),size(tal(i)%std,2)/)
      call h5screate_simple_f(2,dim2,dataspace_id,error)
      call h5dcreate_f(hdfile,"tally_"//trim(talnum)//"/std",H5T_NATIVE_DOUBLE &
     &                 ,dataspace_id,dataset_id,error)
      call h5dwrite_f(dataset_id,H5T_NATIVE_DOUBLE,tal(i)%std,dim2,error)
      call h5sclose_f(dataspace_id,error)
      call h5dclose_f(dataset_id,error)

      ! only write energy edges if a user tally
      if (.not.tal(i)%flux_tally) then

        ! write tally data to file
        dim1 = (/size(tal(i)%E)/)
        call h5screate_simple_f(1,dim1,dataspace_id,error)
        call h5dcreate_f(hdfile,"tally_"//trim(talnum)//"/E",H5T_NATIVE_DOUBLE,&
       &                 dataspace_id,dataset_id,error)
        call h5dwrite_f(dataset_id,H5T_NATIVE_DOUBLE,tal(i)%E,dim1,error)
        call h5sclose_f(dataspace_id,error)
        call h5dclose_f(dataset_id,error)

      end if

      ! close the group
      call h5gclose_f(group_id,error)

    end do

    ! close the file
    call h5fclose_f(hdfile,error)

  end subroutine write_output

!===============================================================================
! GET_TODAY
!> @brief calculates information about date/time of run
!===============================================================================

  subroutine get_today(today_date, today_time)

    character(10), intent(out) :: today_date
    character(8),  intent(out) :: today_time
```

```
    integer        :: val(8)
    character(8)   :: date_
    character(10)  :: time_
    character(5)   :: zone

    call date_and_time(date_, time_, zone, val)
    ! val(1) = year (YYYY)
    ! val(2) = month (MM)
    ! val(3) = day (DD)
    ! val(4) = timezone
    ! val(5) = hours (HH)
    ! val(6) = minutes (MM)
    ! val(7) = seconds (SS)
    ! val(8) = milliseconds

    if (val(2) < 10) then
       if (val(3) < 10) then
          today_date = date_(6:6) // "/" // date_(8:8) // "/" // date_(1:4)
       else
          today_date = date_(6:6) // "/" // date_(7:8) // "/" // date_(1:4)
       end if
    else
       if (val(3) < 10) then
          today_date = date_(5:6) // "/" // date_(8:8) // "/" // date_(1:4)
       else
          today_date = date_(5:6) // "/" // date_(7:8) // "/" // date_(1:4)
       end if
    end if
    today_time = time_(1:2) // ":" // time_(3:4) // ":" // time_(5:6)

  end subroutine get_today

end module output
```

C.5 Particle

```
!=============================================================================!
! MODULE: particle
!
!> @author Bryan Herman
!>
!> @brief Contains information about the particle that is transporting
!=============================================================================!

module particle
```

```
  implicit none
  private
  public :: init_particle

  type, public :: particle_type

    real(8) :: E       ! particle's energy
    logical :: alive   ! am i alive?
    integer :: region  ! material location
    integer :: isoidx  ! isotope index in region
    integer :: reactid ! reaction id

  end type particle_type

contains

!===============================================================================
! INIT_PARTICLE
!> @brief routine to initialize a particle
!===============================================================================

  subroutine init_particle(this)

    ! formal variables
    type(particle_type) :: this ! a particle

    ! initialize
    this%E = 0.0_8
    this%alive = .true.
    this%region = 1
    this%isoidx = 0
    this%reactid = 0

  end subroutine init_particle

end module particle
```

C.6 Physics

```
!===============================================================================!
! MODULE: physics
!
!> @author Bryan Herman
!>
```

```
!> @brief Contains routines to model the physics of the problem
!==============================================================!

module physics

  implicit none
  private
  public :: sample_source,perform_physics,get_eidx

contains

!==============================================================
! SAMPLE_SOURCE
!> @brief routine to sample source from cdf
!==============================================================

  subroutine sample_source()

    use global, only: mat,neut

    ! local variables
    integer :: idx ! index for sampling
    real(8) :: rn  ! sampled random number

    ! sample a random number
    rn = rand(0)

    ! compute index in cdf
    idx = ceiling(rn / mat(1)%source%cdf_width) + 1

    ! bounds checker
    if (idx > size(mat(1)%source%E)) then
      write(*,*) 'Bounds error on source samplings'
      write(*,*) 'Random number:',rn
      write(*,*) 'Index Location:',idx
      stop
    end if

    ! extract that E and set it to neutron
    neut%E = mat(1)%source%E(idx)

  end subroutine sample_source

!==============================================================
! PERFORM_PHYSICS
!> @brief high level routine to perform transport physics
!==============================================================
```

```
  subroutine perform_physics()

    use global, only: neut,add_to_tallies,n_abs,n_fiss

    ! sample region
    neut%region = sample_region()

    ! sample isotope
    neut%isoidx = sample_isotope(neut%region)

    ! a collision has now occurred in a region at an energy, add to tally
    call add_to_tallies()

    ! sample reaction in isotope
    neut%reactid = sample_reaction(neut%region,neut%isoidx)

    ! perform reaction
    if (neut%reactid == 1 .or. neut%reactid == 2) then ! absorption
      neut%alive = .FALSE.
      if (neut%reactid == 1) n_fiss = n_fiss+1
      n_abs = n_abs + 1
    else if (neut%reactid == 3) then ! scattering
      call elastic_scattering(neut%region,neut%isoidx)
    else
      write(*,*) "Something is wrong after isotope sampling"
      stop
    end if

  end subroutine perform_physics

!===============================================================================
! GET_EIDX
!> @brief function to compute the index in unionized energy grid
!===============================================================================

  function get_eidx(E) result(eidx)

    use global, only: mat

    ! formal variables
    real(8)              :: E    ! neutron's energy
    integer              :: eidx ! the energy index

    ! compute index
    eidx = ceiling((log10(E) - log10(mat(1)%E_min))/mat(1)%E_width) + 1
```

```
    ! check bounds
    if (eidx == 0 .or. eidx >=mat(1)%npts) then
      write(*,*) 'Energy index out of bounds!'
      write(*,*) 'Energy:',E
      write(*,*) 'Width:',mat(1)%E_width
      stop
    end if

  end function get_eidx

!=======================================================================
! SAMPLE_REGION
!> @brief function to sample region where interaction occurs
!=======================================================================

  function sample_region() result(region)

    use global, only: n_materials,Dancoff,res_iso,radius,mat,eidx,neut

    ! formal variables
    integer :: region ! region of interaction

    ! local variables
    real(8) :: Pff    ! fuel-to-fuel collision probability
    real(8) :: Pfm    ! fuel-to-moderator collision probability
    real(8) :: Pmf    ! moderator-to-fuel collision probability
    real(8) :: A      ! A factor
    real(8) :: a1     ! alpha 1 factor
    real(8) :: a2     ! alpha 2 factor
    real(8) :: b      ! beta factor
    real(8) :: sig_e  ! macro escape cross section
    real(8) :: sig_t  ! macro total cross section of resonant isotope
    real(8) :: rn     ! sampled random number

    ! set region number
    region = 1

    ! check for more than 1 material
    if (n_materials == 2) then

      ! calculate A
      A = (1.0_8-Dancoff)/Dancoff

      ! calculate alpha 1
      a1 = ((5._8*A+6._8)-sqrt(A**2+36._8*A+36._8))/(2._8*(A+1._8))

      ! calculate alpha 2
```

```
      a2 = ((5._8*A+6._8)+sqrt(A**2+36._8*A+36._8))/(2._8*(A+1._8))

      ! calculate beta
      b = (((4._8*A+6._8)/(A+1._8)) - a1)/(a2 - a1)

      ! calculate macro escape cross section
      sig_e = 1._8/(2._8*radius)

      ! get macro total cross section of resonant isotope
      sig_t = sum(mat(1)%totalxs(eidx,:))

      ! compute fuel-to-fuel collision probability (Carlviks two-term)
      Pff = (b*sig_t)/(a1*sig_e + sig_t) + ((1-b)*sig_t)/(a2*sig_e + sig_t)

      ! compute fuel-to-moderator collision probability
      Pfm = 1._8 - Pff

      ! using reciprocity compute moderator-to-fuel collision probability
      Pmf = Pfm * (sum(mat(1)%totalxs(eidx,:))*mat(1)%vol) /                    &
     &            (sum(mat(2)%totalxs(eidx,:))*mat(2)%vol)

      ! sample random number
      rn = rand(0)

      ! figure out what region currently in and sample accordingly
      if (neut%region == 1) then
        if (rn < Pfm) then
          region = 2
        else
          region = 1
        end if
      else if (neut%region == 2) then
        if (rn < Pmf) then
          region = 1
        else
          region = 2
        end if
      else
        write(*,*) 'Cant find neutron!'
        stop
      end if

    end if
!print *,'Pff:',Pff,'Pfm:',Pfm,'Pmf:',Pmf,'Energy:',neut%E,sig_t,sum(mat(2)%↩
   totalxs(eidx,:))
!read*
  end function sample_region
```

```
!===============================================================================
! SAMPLE_ISOTOPE
!> @brief function to sample interaction isotope
!===============================================================================

  function sample_isotope(region) result(isoidx)

    use global, only: mat,eidx

    ! formal variables
    integer :: region  ! region of interaction
    integer :: isoidx  ! the index of the isotope sampled

    ! local variables
    real(8), allocatable :: pmf(:)  ! probability mass function
    real(8), allocatable :: cdf(:)  ! cumulative distribution function
    real(8)              :: rn      ! sampled random number
    integer              :: i       ! iteration counter

    ! allocate pmf and cdf
    if(.not. allocated(pmf)) allocate(pmf(mat(region)%nisotopes+1))
    if(.not. allocated(cdf)) allocate(cdf(mat(region)%nisotopes+1))

    ! set both to zero
    pmf = 0.0_8
    cdf = 0.0_8

    ! create pmf at that energy index
    pmf(2:size(pmf)) = mat(region)%totalxs(eidx,:) /                          &
   &                   sum(mat(region)%totalxs(eidx,:))

    ! create cdf from pmf
    do i = 1,size(pmf)
      cdf(i) = sum(pmf(1:i))
    end do

    ! sample random number
    rn = rand(0)

    ! do linear table search on cdf to find which isotope
    do i = 1,size(cdf)
      if (rn <= cdf(i)) then
        isoidx = i - 1
        exit
      end if
    end do
```

```
    ! check iso
    if (isoidx == 0) then
      isoidx = 1
    end if

    ! deallocate pmf and cdf
    if (allocated(pmf)) deallocate(pmf)
    if (allocated(cdf)) deallocate(cdf)

  end function sample_isotope

!=======================================================================
! SAMPLE_REACTION
!> @brief function to sample reaction type
!=======================================================================

  function sample_reaction(region,isoidx) result(reactid)

    use global, only: mat,eidx

    ! formal variables
    integer :: region   ! region of interaction
    integer :: isoidx   ! the sampled isotope index
    integer :: reactid  ! the id of the reaction type

    ! local variables
    real(8) :: pmf(4)  ! probability mass function
    real(8) :: cdf(4)  ! cumulative distribution function
    real(8) :: rn      ! sampled random number
    integer :: i       ! iteration counter

    ! set up pmf
    pmf = (/0.0_8,mat(region)%isotopes(isoidx)%xs_fiss(eidx),                &
   &              mat(region)%isotopes(isoidx)%xs_capt(eidx),                &
                  mat(region)%isotopes(isoidx)%xs_scat(eidx)/)

    ! normalize pmf
    pmf = pmf / sum(pmf)

    ! compute cdf
    do i = 1,4
      cdf(i) = sum(pmf(1:i))
    end do

    ! sample random number
    rn = rand(0)
```

```
    ! perform linear table search
    do i = 1,4
      if (rn < cdf(i)) then
        reactid = i - 1
        exit
      end if
    end do

  end function sample_reaction

!=======================================================================
! ELASTIC_SCATTERING
!> @brief routine to perform thermal/asymptotic elastic scattering physics
!=======================================================================

  subroutine elastic_scattering(region,isoidx)

    use global, only: neut,mat,kT

    ! formal variables
    integer :: region ! region of interaction
    integer :: isoidx ! isotope sampled index

    ! local variables
    integer :: i      ! iteration counter
    integer :: idx    ! index in cdf vector
    integer :: kTidx  ! index in kT vector
    real(8) :: rn     ! sampled random number
    real(8) :: EkT    ! energy / kT
    real(8) :: Eint   ! interpolated E value
    real(8), allocatable :: Evec(:)

    ! sample random number
    rn = rand(0)

    ! check for thermal scattering
    if (neut%E < 4e-6_8 .and. mat(region)%isotopes(isoidx)%thermal) then

      ! get index in cdf
      idx = ceiling(rn/mat(region)%isotopes(isoidx)%thermal_lib%cdf_width)

      ! check index
      if (idx == 0) idx = 1

      ! preallocate energy vector
      if (.not.allocated(Evec))                                          &
```

```
     & allocate(Evec(size(mat(region)%isotopes(isoidx)%thermal_lib%kTvec)))

      ! set possible energy ratios vector
      Evec = mat(region)%isotopes(isoidx)%thermal_lib%Erat(idx,:)

      ! get energy in kT units
      EkT = neut%E/kT

      ! find index in kT space
      do i = 1,size(mat(region)%isotopes(isoidx)%thermal_lib%kTvec)
        if (EkT < mat(region)%isotopes(isoidx)%thermal_lib%kTvec(i)) then
          kTidx = i
          exit
        end if
      end do

      ! interpolate on energy value
      if (kTidx == 1) then
        neut%E = Evec(kTidx)
      else
        ! perform linear interplation on kT value
        Eint = Evec(kTidx-1) + (EkT -                                          &
       &mat(region)%isotopes(isoidx)%thermal_lib%kTvec(kTidx-1))*((Evec(kTidx) &
       &- Evec(kTidx-1))/(mat(region)%isotopes(isoidx)%thermal_lib%kTvec(kTidx)&
       &- mat(region)%isotopes(isoidx)%thermal_lib%kTvec(kTidx-1)))

       ! multiply by incoming energy
       neut%E = neut%E*Eint

      end if

      ! deallocate energy vector
      if(allocated(Evec)) deallocate(Evec)

    else

      ! perform asymptotic elastic scattering
      neut%E = neut%E - neut%E*(1-mat(region)%isotopes(isoidx)%alpha)*rn;

    end if

  end subroutine elastic_scattering

end module physics
```

C.7 Tally

```
!=======================================================================!
! MODULE: tally
!
!> @author Bryan Herman
!>
!> @brief Contains information about tallying quantities
!=======================================================================!

module tally

  implicit none
  private
  public :: set_spectrum_tally,add_to_tally,bank_tally,deallocate_tally,       &
 &          set_user_tally,calculate_statistics,set_kinf_tally

  type, public :: tally_type

    real(8), allocatable :: E(:)       ! user defined energy structure
    real(8), allocatable :: val(:,:)     ! the temporary value
    real(8), allocatable :: sum(:,:)     ! the sum for the mean and var
    real(8), allocatable :: sum_sq(:,:) ! the sum for the variable
    real(8), allocatable :: mean(:,:)    ! mean of tallies
    real(8), allocatable :: std(:,:)     ! standard deviation of tallies
    logical :: flux_tally = .false.    ! is this the flux tally
    integer :: nbins                   ! number of tally regions
    real(8) :: width                   ! the uniform width
    real(8) :: emax                    ! max e
    real(8) :: emin                    ! min e
    integer :: react_type              ! reaction type id
    integer :: isotope                 ! isotope number
    integer :: region                  ! region for micros
    logical :: dv = .false.            ! divide by volume of regions

  end type tally_type

contains

!=======================================================================
! SET_USER_TALLY
!> @brief routine to intialize user-defined tallies
!=======================================================================

  subroutine set_user_tally(this,Ebins,n,react_type,isotope,region,n_materials,&
                            dv)

    ! formal variables
    type(tally_type) :: this        ! a tally
```

```
    integer           :: n           ! size of Ebins
    integer           :: react_type  ! reaction type
    integer           :: isotope     ! isotope for multiplier
    integer           :: region      ! region filter for isotope
    integer           :: n_materials ! number of material tally regions
    real(8)           :: Ebins(n)    ! vector of energy bins
    logical           :: dv          ! divide by volume

    ! preallocate user-defined energy structure
    if (.not.allocated(this%E)) allocate(this%E(n))

    ! set divide by volume
    this%dv = dv

    ! set energy structure
    this%E = Ebins

    ! set reaction type
    this%react_type = react_type

    ! set isotope
    this%isotope = isotope

    ! set region
    this%region = region

    ! preallocate vectors
    if(.not.allocated(this%val)) allocate(this%val(n-1,n_materials))
    if(.not.allocated(this%sum)) allocate(this%sum(n-1,n_materials))
    if(.not.allocated(this%sum_sq)) allocate(this%sum_sq(n-1,n_materials))

    ! preallocate mean and stdev
    if (.not.allocated(this%mean)) allocate(this%mean(n-1,n_materials))
    if (.not.allocated(this%std))  allocate(this%std(n-1,n_materials))

    ! zero out tallies
    this%val = 0.0_8
    this%sum = 0.0_8
    this%sum_sq = 0.0_8

  end subroutine set_user_tally

!===============================================================================
! SET_SPECTRUM_TALLY
!> @brief routine to initialize all tallies
!===============================================================================
```

```
  subroutine set_spectrum_tally(this,emax,emin,n_materials)

    ! formal variables
    type(tally_type) :: this          ! a tally
    integer          :: n_materials   ! number of materials
    real(8)          :: emax          ! max e
    real(8)          :: emin          ! min e

    ! set up automatic flux tally
    this%flux_tally = .true.
    this%nbins = 5000
    this%emax = emax
    this%emin = emin
    this%width = (log10(emax) - log10(emin))/dble(this%nbins)

    ! preallocate vectors
    if(.not.allocated(this%val)) allocate(this%val(5000,n_materials))
    if(.not.allocated(this%sum)) allocate(this%sum(5000,n_materials))
    if(.not.allocated(this%sum_sq)) allocate(this%sum_sq(5000,n_materials))

    ! preallocate mean and stdev
    if (.not.allocated(this%mean)) allocate(this%mean(5000,n_materials))
    if (.not.allocated(this%std))  allocate(this%std(5000,n_materials))

    ! zero out tallies
    this%val = 0.0_8
    this%sum = 0.0_8
    this%sum_sq = 0.0_8

  end subroutine set_spectrum_tally

!===============================================================================
! SET_KINF_TALLY
!> @brief routine to initiaize kinf nu-fission tally
!===============================================================================

  subroutine set_kinf_tally(this,emax,emin,n_materials)

    ! formal variables
    type(tally_type) :: this          ! a tally
    integer          :: n_materials   ! number of materials
    real(8)          :: emax          ! max e
    real(8)          :: emin          ! min e

    ! preallocate user-defined energy structure
    if (.not.allocated(this%E)) allocate(this%E(2))
```

```
    ! set energy structure
    this%E(1) = emin
    this%E(2) = emax

    ! set reaction type
    this%react_type = 4

    ! preallocate vectors
    if(.not.allocated(this%val)) allocate(this%val(1,n_materials))
    if(.not.allocated(this%sum)) allocate(this%sum(1,n_materials))
    if(.not.allocated(this%sum_sq)) allocate(this%sum_sq(1,n_materials))

    ! preallocate mean and stdev
    if (.not.allocated(this%mean)) allocate(this%mean(1,n_materials))
    if (.not.allocated(this%std))  allocate(this%std(1,n_materials))

    ! zero out tallies
    this%val = 0.0_8
    this%sum = 0.0_8
    this%sum_sq = 0.0_8

  end subroutine set_kinf_tally

!===============================================================================
! ADD_TO_TALLY
!> @brief routine to add quantities during transport of a particle
!===============================================================================

  subroutine add_to_tally(this,fact,totxs,E,region)

    ! formal variables
    type(tally_type) :: this     ! a tally
    integer          :: region   ! region id
    real(8)          :: fact     ! multiplier for tally
    real(8)          :: totxs    ! totalxs
    real(8)          :: E        ! neutron energy

    ! local variables
    integer :: i      ! iteration counter
    integer :: idx=0  ! index in tally grid

    ! use uniform grid sampling if flux tally
    if (this%flux_tally) then

      ! calculate index
      idx = ceiling((log10(E) - log10(this%emin))/this%width)
```

```
    else

      ! check for output bounds
      if (E < minval(this%E) .or. E > maxval(this%E)) then
        print *,'energy out of tally bounds'
        return
      end if

      ! begin loop around energy vector to get index
      do i = 1,size(this%E)
        if (E < this%E(i)) then
          idx = i - 1
          exit
        end if
      end do

    end if

    ! add to tally
    if (idx /= 0) this%val(idx,region) = this%val(idx,region) + fact/totxs

  end subroutine add_to_tally

!===============================================================================
! BANK_TALLY
!> @brief routine to bank a histories tallies
!===============================================================================

  subroutine bank_tally(this)

    ! formal variables
    type(tally_type) :: this ! a tally

    ! record to sums
    this%sum    = this%sum    + this%val
    this%sum_sq = this%sum_sq + this%val**2

    ! zero out temp value
    this%val = 0.0_8

  end subroutine bank_tally

!===============================================================================
! CALCULATE_STATISTICS
!> @brief routine to compute mean and standard deviation of tallies
!===============================================================================
```

```
  subroutine calculate_statistics(this,n)

    ! formal variables
    type(tally_type) :: this ! a tally
    integer          :: n    ! number of histories run

    ! compute mean
    this%mean =  this%sum / dble(n)

    ! compute standard deviation
    this%std = sqrt((this%sum_sq/dble(n) - this%mean**2)/dble(n-1))

  end subroutine calculate_statistics

!===============================================================================
! DEALLOCATE_TALLY
!> @brief routine to deallocate tally types
!===============================================================================

  subroutine deallocate_tally(this)

    ! formal variables
    type(tally_type) :: this ! a tally

    ! deallocate all
    if (allocated(this%E)) deallocate(this%E)
    if (allocated(this%val)) deallocate(this%val)
    if (allocated(this%sum)) deallocate(this%sum)
    if (allocated(this%sum_sq)) deallocate(this%sum_sq)

  end subroutine deallocate_tally

end module tally
```