Problem Set 3 Solutions

22.211 Reactor Physics I

Due: 10 March 2012

Bryan Herman

Problem: Add U-238 resonance absorption model to your slowing down MC code with the following:

- Model absorption resonances from 0 to 1 keV (ignore resonance scattering)
- Use 0.1 barns for U-238 capture cross section above 1 keV
- Explicitly include SLBW model for the lowest 14 s-wave resonances
- Generate simple "statistical model" for resonance up to l keV
- Assume coolant temperature is always 300 K and U-238 temperature is specified
- Follow neutrons down to 1e-5 eV cutoff
- Add pure $1/v$ absorber in at 2 barns/Hydrogen atom

Self-shielding: Below are the spectrum plots and effective resonance integrals. All results generated with 1 million histories.

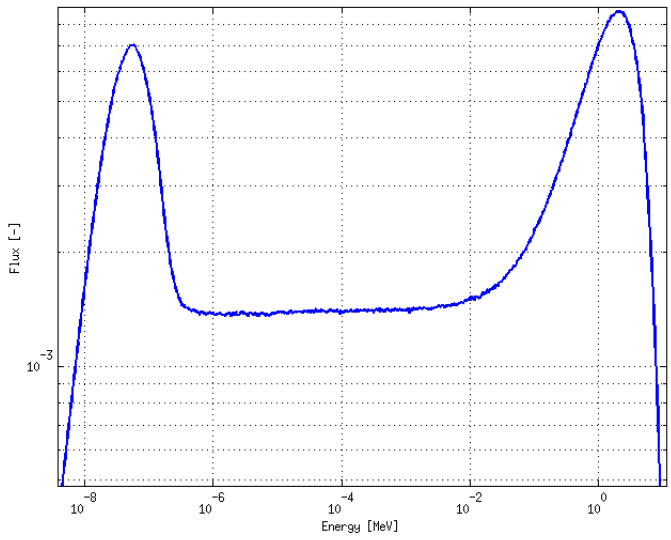

Figure 1: Flux Spectrum - U/H 1e-4%

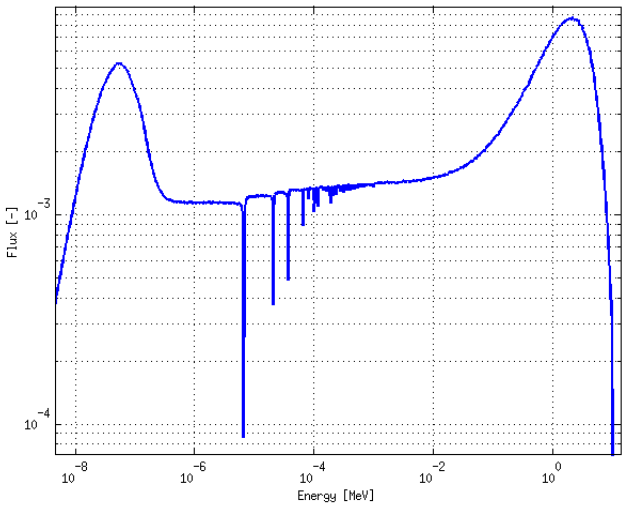

Figure 2: Flux Spectrum - U/H 10%

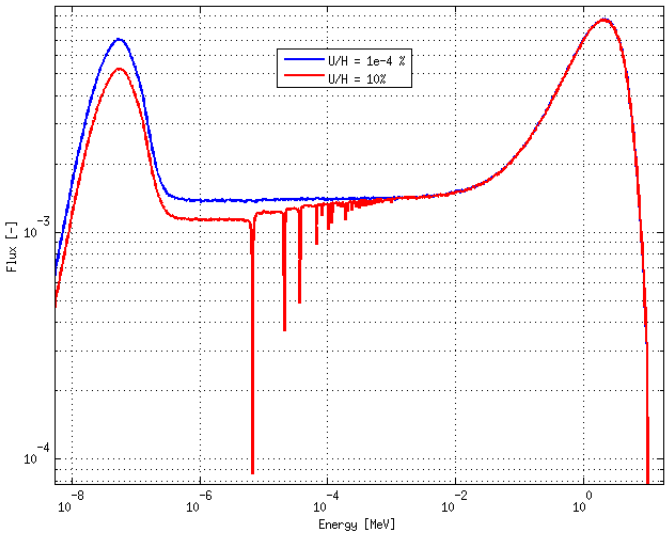

Figure 3: Flux Spectrum - U/H Comparison

Table 1: Effective Resonance Integrals - U-238 @ 300K

Energy Range [eV]	U/H = 1×10^{-6}	U/H = 0.10
0 - 1	21.24	20.90
1 - 6	1.47	1.46
6 - 10	127.00	13.83
10 - 25	66.57	6.93
25 - 50	41.07	5.24
50 - 100	12.24	2.87
100 - 1000	23.28	10.47

Doppler: Below are effective resonance integral results. All results generated with 1 million histories.

Table 2: Effective Resonance Integrals - U-238 @ 300K

Energy Range [eV]	U/H = 1%	U/H = 10%	U/H = 100%
0 - 1	21.21	20.90	18.42
1 - 6	1.47	1.46	1.41
6 - 10	49.26	13.83	4.06
10 - 25	25.36	6.93	2.16
25 - 50	17.95	5.24	1.63
50 - 100	8.02	2.87	0.82
100 - 1000	19.28	10.47	3.27

Table 3: Effective Resonance Integrals - U-238 @ 600K

Energy Range [eV]	U/H = 1%	U/H = 10%	U/H = 100%
0 - 1	21.21	20.90	18.41
1 - 6	1.47	1.46	1.41
6 - 10	55.42	15.08	4.14
10 - 25	28.78	7.60	2.21
25 - 50	20.06	5.86	1.66
50 - 100	8.73	3.31	0.90
100 - 1000	20.01	11.60	3.77

Table 4: Effective Resonance Integrals - U-238 @ 900K

Energy Range [eV]	U/H = 1%	U/H = 10%	U/H = 100%
0 - 1	21.21	20.90	18.41
1 - 6	1.47	1.47	1.42
6 - 10	59.67	16.16	4.22
10 - 25	31.08	8.19	2.25
25 - 50	31.52	6.06	1.69
50 - 100	9.14	3.62	0.96
100 - 1000	20.47	12.31	4.14

Table 5: Effective Resonance Integrals - U-238 @ 1200K

Energy Range [eV]	U/H = 1%	U/H = 10%	U/H = 100%
0 - 1	21.21	20.90	18.42
1 - 6	1.48	1.47	1.42
6 - 10	63.03	17.14	4.31
10 - 25	32.91	8.72	2.30
25 - 50	22.59	6.41	1.73
50 - 100	9.44	3.86	1.01
100 - 1000	20.74	12.84	4.46

Below are a set of 3 tables comparing the effective resonance integrals of 600K, 900K and 1200K with 300K. This will allow us to see the doppler effect on different resonances.

Table 6: Percent Difference of Effective Resonance Integrals - U-238 @ 600K (wrt to 300K)

Energy Range [eV]	U/H = 1%	U/H = 10%	U/H = 100%
0 - 1	0	0	-0.05
1 - 6	0	0	0
6 - 10	12.51	9.04	1.97
10 - 25	13.49	9.67	2.31
25 - 50	11.75	11.83	1.84
50 - 100	8.85	15.33	9.75
100 - 1000	3.78	10.79	15.29

Table 7: Percent Difference of Effective Resonance Integrals - U-238 @ 900K (wrt to 300K)

Energy Range [eV]	U/H = 1%	U/H = 10%	U/H = 100%
0 - 1	0	0	-0.05
1 - 6	0	0.68	0.71
6 - 10	21.13	16.85	3.94
10 - 25	22.56	18.18	4.17
25 - 50	75.60	15.65	3.68
50 - 100	13.97	26.13	17.07
100 - 1000	6.17	17.57	26.61

Table 8: Percent Difference of Effective Resonance Integrals - U-238 @ 1200K (wrt to 300K)

Energy Range [eV]	U/H = 1%	U/H = 10%	U/H = 100%
0 - 1	0	0	0
1 - 6	0.68	0.68	0.71
6 - 10	27.95	23.93	6.16
10 - 25	29.77	25.83	6.48
25 - 50	25.85	22.33	6.14
50 - 100	17.71	34.49	23.17
100 - 1000	7.57	22.64	36.39

A Sample Input File for Slowing Down Code

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

<!-- Settings Information -->
  <settings>
    <histories> 1000000 </histories>
    <seed> 5 </seed>
    <source_type> 1 </source_type>
    <source_path> /home/bherman/Documents/Spring2012/211/SlowMC/lib/fission.h5 </source_pat
    <Dancoff> 0 </Dancoff>
    <res_iso> U-238 </res_iso>
  </settings>


  <materials>
    <material>
      <type> homogeneous </type>
      <nuclide name="H-1" N="1" A="1" thermal="true" >
        <path> /home/bherman/Documents/Spring2012/211/SlowMC/lib/H_1.h5 </path>
      </nuclide>
      <nuclide name="1/v" N="1" A="1" thermal="false" >
        <path> /home/bherman/Documents/Spring2012/211/SlowMC/lib/v_abs.h5 </path>
      </nuclide>
      <nuclide name="U-238" N="0.1" A="238" thermal="false" >
        <path> /home/bherman/Documents/Spring2012/211/SlowMC/lib/U_238_300.h5 </path>
      </nuclide>
    </material>
  </materials>

  <tallies>
    <tally type="micro_capture" isotope="3" >
      <Ebins> 1e-11 1e-6 6e-6 10e-6 25e-6 50e-6 100e-6 1000e-6 </Ebins>
    </tally>
    <tally type="flux" >
      <Ebins> 1e-11 1e-6 6e-6 10e-6 25e-6 50e-6 100e-6 1000e-6 </Ebins>
    </tally>
  </tallies>

</input>
```

B Main Codes

B.1 SLBW Code

```
% Single Level Breit Wigner xs generator for U-238
tic
```

```
% Inputs for user
isoname = 'U_238';  % isotope name
n_res = 14;         % number of resonances to read from file
T = 1200;           % temperature of resonances
sig_pot = 11.2934; % potential cross section of isotope
A = 238;            % isotope atomic weight
maxE = 1000;        % max energy in eV

% load isotope file
load('U_238_res.txt');

% constants
k = 8.6173324e-5;

% extract resonance information from loaded dataa
Gg = U_238_res(1:n_res,4);
Gn = U_238_res(1:n_res,3);
E0 = U_238_res(1:n_res,1);

% create energy vector
dE = 0.001;
E = 10.^(log10(1e-5):dE:log10(20e6))';
sizeE = length(E);

% set psi-chi vectors
psi = zeros(sizeE,1);
chi = zeros(sizeE,1);

% initialize xs
xs = zeros(sizeE,2);
xs(:,2) = sig_pot;

% begin loop around resonances
for j = 1:n_res

    % psi-chi parameters
    G = Gg(j) + Gn(j);
    r = 2603911/E0(j)*((A+1)/A);
    q = sqrt(r*sig_pot);
    xi = G*sqrt(A/(4*k*T*E0(j)));
    x = 2*(E-E0(j))/G;

    % compute psi-chi functions
    y = ((x+1i)/2*xi);
    psichi = pi*xi/(2*sqrt(pi))*W(y); % compute complex value
    psi = real(psichi);
    chi = 2*imag(psichi);

    % compute xs
    xs(:,1) = (Gn(j)/G)*(Gg(j)/G)*sqrt(E0(j)./E).*(r*psi) + xs(:,1);
    xs(:,2) = (Gn(j)/G)*(Gn(j)/G)*(r*psi + q*chi) + xs(:,2);

    % display resonance info
    fprintf('Completed Resonance: %d\n',E0(j));
```

```
end

% append pseudo resonances (25eV spacing)
Elast = E0(length(E0));
clear E0;
E0 = Elast + 25;
Gg = 0.023;

% begin loop around building pseudo resonances
while E0 < maxE

    % compute neutron width
    Gn = 0.050*sqrt(E0/Elast);

    % psi-chi parameters
    G = Gg + Gn;
    r = 2603911/E0*((A+1)/A);
    q = sqrt(r*sig_pot);
    xi = G*sqrt(A/(4*k*T*E0));
    x = 2*(E-E0)/G;

    % compute psi-chi functions
    y = ((x+1i)/2*xi);
    psichi = pi*xi/(2*sqrt(pi))*W(y); % compute complex value
    psi = real(psichi);
    chi = imag(psichi);

    % compute xs
    xs(:,1) = (Gn/G)*(Gg/G)*sqrt(E0./E).*(r*psi) + xs(:,1);
    xs(:,2) = (Gn/G)*(Gn/G)*(r*psi + q*chi) + xs(:,2);

    % get next E0
    E0 = E0 + 25;

    % display resonance info
    fprintf('Completed Resonance: %d\n',E0);

end
toc

% change units on E
E = E/1e6;

% zero out xs over 1 keV
xs(:,1) = xs(:,1).*(E <= 1e-3);

% put 0.1 barns after
xs(:,1) = xs(:,1) + (E > 1e-3)*0.1;

% get size
sizeE = length(xs(:,1));

% get capture
```

```
xs_capt = xs(:,1);

% set scattering to 0
xs_scat = zeros(sizeE,1);

% filename
hdfile = horzcat(isoname,'_',num2str(T),'.h5');

% write out hdf5 file
delete(hdfile);
h5create(hdfile,'/vecsize',1);
h5write(hdfile,'/vecsize',sizeE);
h5create(hdfile,'/xs_scat',sizeE);
h5write(hdfile,'/xs_scat',xs_scat);
h5create(hdfile,'/xs_capt',sizeE);
h5write(hdfile,'/xs_capt',xs_capt);
h5create(hdfile,'/E_width',1);
h5write(hdfile,'/E_width',dE);
```

B.2 Thermal Scattering Kernel Generation Code

```
function [thermalcdf,Erat]  = free_gas(A,T,sizeN,kTfactor)
% free gas thermal scattering kernel generation

% constants
k = 8.6173324e-5;
eta = (A+1)/(2*sqrt(A));
rho = (A-1)/(2*sqrt(A));
alpha = ((A-1)/(A+1))^2;

% create output space
p = zeros(sizeN,length(kTfactor));
thermalcdf = zeros(sizeN,length(kTfactor));

% begin loop around temperatures
for i = 1:length(kTfactor)

    % create energy space
    kT = k*T;
    E = kT*kTfactor(i);
    Ep = linspace(0,5*E,sizeN);

    % compute probability
    p(:,i) =((eta^2/2)*(erf(eta*sqrt(Ep/kT) - rho*sqrt(E/kT)) - ...
        erf(eta*sqrt(Ep/kT) + rho*sqrt(E/kT)) + ...
        exp((E-Ep)/kT).*(erf(eta*sqrt(E/kT) - rho*sqrt(Ep/kT)) + ...
        erf(eta*sqrt(E/kT) + rho*sqrt(Ep/kT))))).*(Ep >= E) + ...
        ((eta^2/2)*(erf(eta*sqrt(Ep/kT) - rho*sqrt(E/kT)) + ...
        erf(eta*sqrt(Ep/kT) + rho*sqrt(E/kT)) + ...
        exp((E-Ep)/kT).*(erf(eta*sqrt(E/kT) - rho*sqrt(Ep/kT)) - ...
        erf(eta*sqrt(E/kT) + rho*sqrt(Ep/kT))))).*(Ep < E);
```

```
    % multiply by alpha
    p(:,i) = p(:,i)*(1-alpha);

    % relative energy ratio
    Erat = Ep/E;

    % loop around energy ratio and integerate
    for j = 2:length(Erat)

        % perform integral
        thermalcdf(j,i) = trapz(Erat(1:j),p(1:j,i));

    end

    % renormalize cdf
    thermalcdf(:,i) = thermalcdf(:,i)/thermalcdf(length(thermalcdf),i);

    % plot pdfs and cdfs
    rn1 = rand(1,1);
    rn2 = rand(1,1);
    rn3 = rand(1,1);
    figure(3)
    xlabel('Scattered Energy Ratio (out/in)')
    ylabel('Probability')
    hold on
    pdfplot = plot(Erat,p(:,i),'Color',[rn1,rn2,rn3],'LineWidth',2.0);
    if i == 1
        legend(horzcat('E = ',num2str(kTfactor(i)),' kT'))
    else
        [LEGH,OBJH,OUTH,OUTM] = legend;
        legend([OUTH;pdfplot],OUTM{:},horzcat('E = ',num2str(kTfactor(i)),' kT'));
    end

    figure(2)
    xlabel('Scattered Energy Ratio (out/in)')
    ylabel('Cumulative Probability')
    hold on
    cdfplot = plot(Erat,thermalcdf(:,i),'Color',[rn1,rn2,rn3],'LineWidth',2.0);
    if i == 1
        legend(horzcat('E = ',num2str(kTfactor(i)),' kT'))
    else
        [LEGH,OBJH,OUTH,OUTM] = legend;
        legend([OUTH;cdfplot],OUTM{:},horzcat('E = ',num2str(kTfactor(i)),' kT'));
    end

end

%% Re-adjust cdf for sampling optimization
Enew = zeros(sizeN,length(kTfactor));
cdfnew = linspace(0,1,sizeN)';
thermalcdftemp = thermalcdf;

% correct thermal cdf for precision unique issues
```

```
for i = 1:size(thermalcdftemp,2)
    for j = 1:size(thermalcdftemp,1)-1
        if abs(thermalcdftemp(j,i) - thermalcdftemp(j+1,i)) < 1e-8
            thermalcdf(j+1,i) = 1.01*thermalcdf(j,i);
        end
    end
end

% create new energy vector
for i = 1:size(thermalcdf,2)
    Enew(:,i) = interp1(thermalcdf(:,i),Erat,cdfnew,'linear');
end

% bank to saved variables
thermalcdf = cdfnew;
Erat = Enew;
width = thermalcdf(2) - thermalcdf(1);
```

B.3 Slowing Down

```
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,add_to_tallies,         &
   &                      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)

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

        ! record collision temp tally
        call add_to_tallies()

        ! perform physics
        call perform_physics()

        ! check for energy cutoff
        if (neut%E < emin) neut%alive = .FALSE.

      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

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

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

    ! 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

        ! absorption
        case(1)
          fact = sum(mat(neut%region)%absorxs(eidx,:))

        ! scattering
        case(2)
          fact = sum(mat(neut%region)%scattxs(eidx,:))
```

```
        ! micro capture
        case(3)
          fact = mat(neut%region)%isotopes(tal(i)%isotope)%xs_capt(eidx)
        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

    ! begin loop over tallies
    do i = 1,n_tallies

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

    end do

  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
    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
real(8), allocatable            :: Ebins(:)     ! tally energy bins

! 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 = 1
else
  n_tallies = size(tallies_%tally) + 1
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 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
      vol = 1.0_8
      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-1

    ! set reaction type
    select case(trim(tallies_%tally(i)%type))
      case('flux')
        react_type = 0
      case('absorption')
        react_type = 1
      case('scattering')
        react_type = 2
      case('micro_capture')
        react_type = 3
        isotope = tallies_%tally(i)%isotope
      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,n_materials↩
         )

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

    end do

    ! set up spectrum tally
    call set_spectrum_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), allocatable   :: xs_capt(:)  ! capture micro xs
    real(8), allocatable   :: xs_scat(:)  ! scattering 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

  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)%alpha = ((A-1)/(A+1))**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))

  ! 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

  ! 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)

  ! 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))

    ! 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%scattxs(:,i) = iso%N*(iso%xs_scat)
      this%absorxs(:,i) = iso%N*(iso%xs_capt)
      this%totalxs(:,i) = iso%N*(iso%xs_capt + iso%xs_scat)

    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)

    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)

  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()

    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
   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 write statements
100 format (1X,A,T35,"= ",ES11.4," seconds")

    ! 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(1,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(1,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

    ! sample region
    neut%region = sample_region()

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

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

    ! perform reaction
    if (neut%reactid == 1) then ! absorption
      neut%alive = .FALSE.
```

```
    else if (neut%reactid == 2) 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 = mat(1)%totalxs(eidx,res_iso)

  ! compute fuel-to-fuel collision probability
  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

  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(3)  ! probability mass function
    real(8) :: cdf(3)  ! 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_capt(eidx),                    &
   &                                 mat(region)%isotopes(isoidx)%xs_scat(eidx)/)

    ! normalize pmf
    pmf = pmf / sum(pmf)

    ! compute cdf
    do i = 1,3
      cdf(i) = sum(pmf(1:i))
    end do

    ! sample random number
    rn = rand(0)

    ! perform linear table search
    do i = 1,3
      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

  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

  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,n_materials)

    ! formal variables
    type(tally_type) :: this         ! a tally
    integer          :: n            ! size of Ebins
    integer          :: react_type   ! reaction type
    integer          :: isotope      ! isotope for multiplier
    integer          :: n_materials ! number of material tally regions
    real(8)          :: Ebins(n)     ! vector of energy bins

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

    ! set energy structure
    this%E = Ebins

    ! set reaction type
    this%react_type = react_type

    ! set isotope
    this%isotope = isotope

    ! 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(size(this%sum),          &
   &                                                  n_materials))
```

```
    if (.not.allocated(this%std))  allocate(this%std(size(this%sum),          &
   &                                                                 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 = 1000
    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(1000,n_materials))
    if(.not.allocated(this%sum)) allocate(this%sum(1000,n_materials))
    if(.not.allocated(this%sum_sq)) allocate(this%sum_sq(1000,n_materials))

    ! preallocate mean and stdev
    if (.not.allocated(this%mean)) allocate(this%mean(size(this%sum),         &
   &                                                                 n_materials))
    if (.not.allocated(this%std))  allocate(this%std(size(this%sum),          &
   &                                                                 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
```

```
!=============================================================================
! 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)) return

      ! 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))

  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
```