Problem Set 2 Solutions

22.211 Reactor Physics I

Due: 29 February 2012

Bryan Herman

Problem 1. Generate the H-1 thermal scattering kernels vs. temperature.

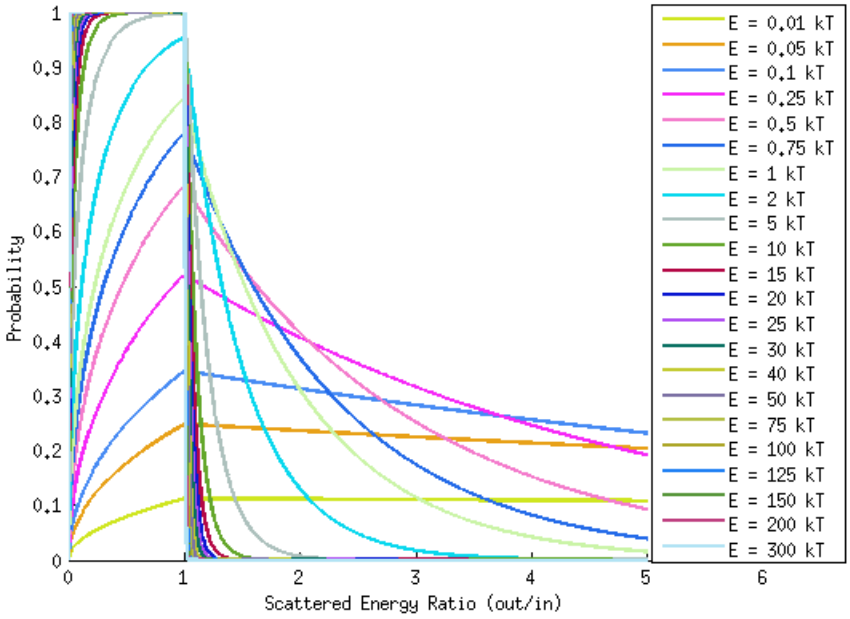

Figure 1: H-1 Thermal Scattering Kernels PDF

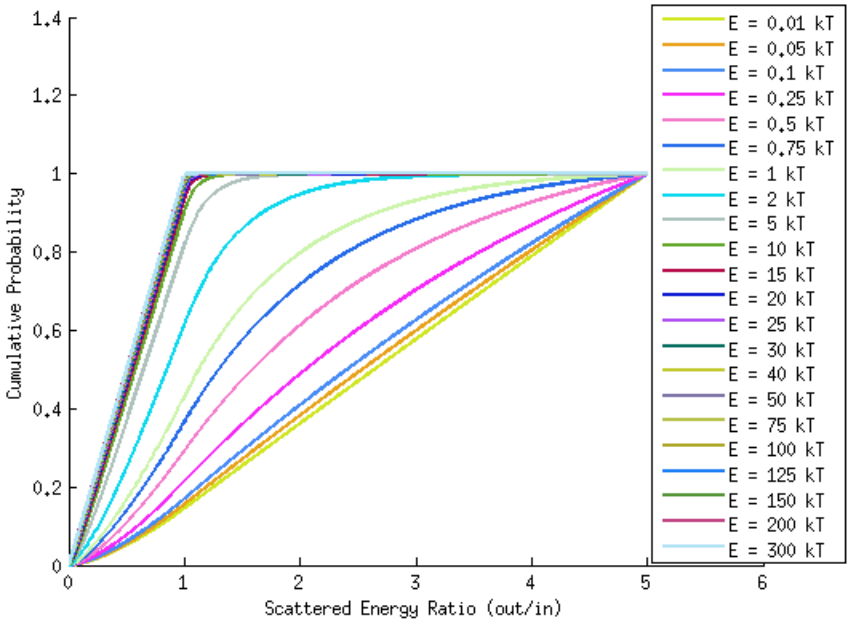

Figure 2: H-1 Thermal Scattering Kernels CDF

Problem 2. Generate the C-12 thermal scattering kernels vs. temperature.

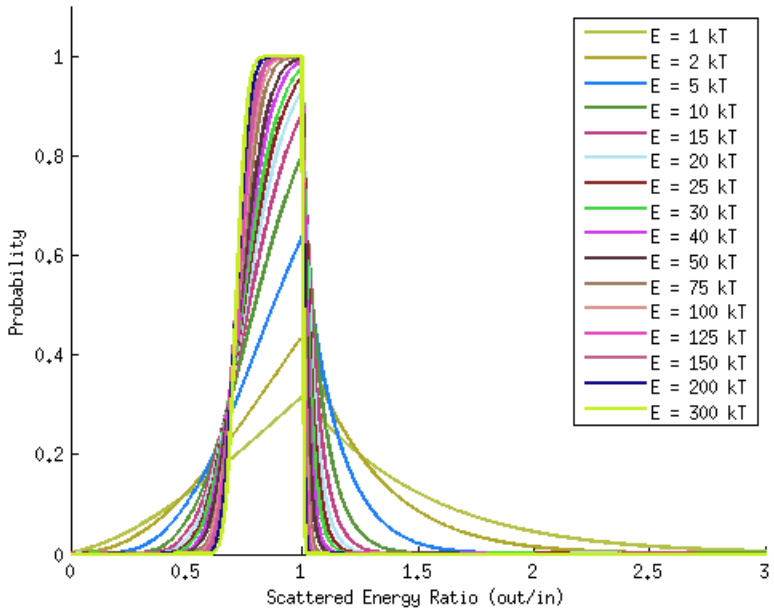

Figure 3: C-12 Thermal Scattering Kernels PDF

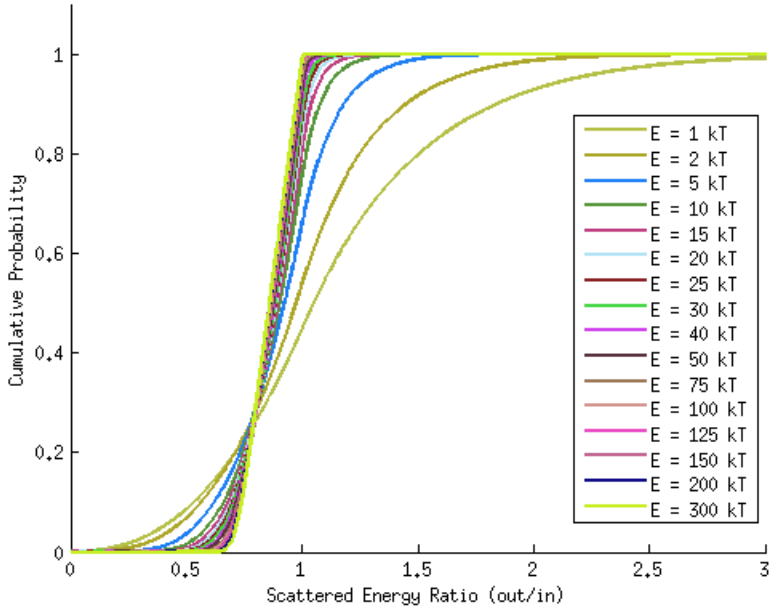

Figure 4: C-12 Thermal Scattering Kernels CDF

Problem 3. Compute the spectrum (flux vs. lethargy) in homogeneous H-1 at 300K. Do the following:

– Random fission neutron emission

– Asymptotic elastic slowing down to 4 eV

– ENDF/B-VII H-1 elastic scattering cross section vs. energy

– Free gas hydrogen thermal scattering below 4 eV

– Artificial $1/v$ absorber with cross section of 7 barns per H-1 atom

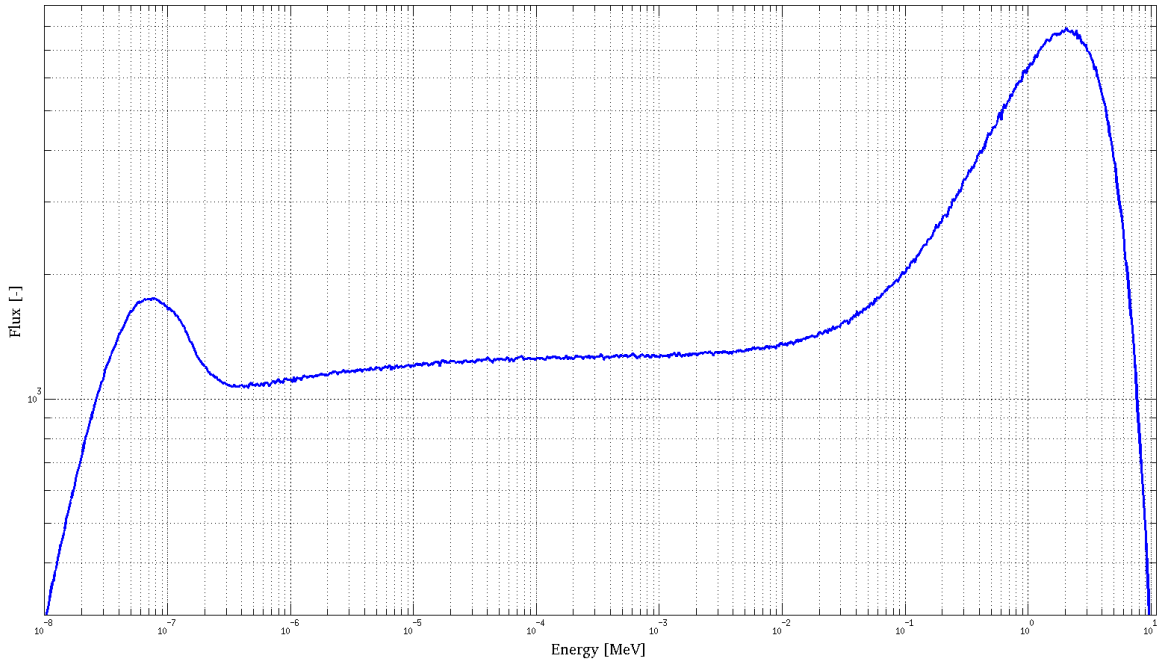

Figure 5: Flux Spectrum with Thermal Scattering and 1/v absorber

Note, in order to get this plot to be smooth with the correct thermal shape, I had to use a spectrum of kT values that were shown above.

Problem 4. Compute the mean number of collisions for neutrons to reach 1.0 eV.

From my code the mean number of collisions for neutrons to reach 1.0 eV is

$$\boxed{14.95.}$$

The analytical formula predicts

$$n = \ln\left(\frac{E_i}{E_f}\right) = \ln\left(\frac{2\times 10^6}{1}\right) = 14.51.$$

Therefore, we are in the ball park of the correct answer.

Problem 5. Compute the mean number of collisions per neutron.

From my code the mean number of collisions for neutrons to reach 1.0 eV is

$$\boxed{21.24.}$$

The analytical formula predicts

$$n = \ln\left(\frac{E_i}{E_f}\right) = \ln\left(\frac{2\times 10^6}{0.0001}\right) = 23.72.$$

We are again in the ball park of the correct answer. Although we are not as close, the analytical formulation does not account for 1/v absorber or thermal scattering.

Problem 6. Compute the mean neutron lifetime.

From my code I computed the mean neutron lifetime as

$$\tau = \frac{1}{v\sigma_t} = 6.52577\times 10^{-5}\,\frac{\mathrm{s}}{\mathrm{m}\cdot\mathrm{b}},$$

where v is the velocity given by

$$v = \sqrt{\frac{2E}{m_n}}$$

and σ_t is the total microscopic cross section given by

$$\sigma_t = \sigma_s + \sigma_a.$$

We still need to multiply the microscopic total cross section by the number density of hydrogen. The number density of hydrogen is

$$N_H = n_H N_{H_2O} = n_H\frac{\rho N_A}{M} = 2\frac{1\cdot 0.6022}{18.01528} = 0.066854\,\frac{1}{\mathrm{cm}\cdot\mathrm{b}}$$

Therefore, the mean neutron lifetime is

$$\boxed{\tau = 9.76\,\mu\mathrm{s}.}$$

This time is smaller than the value in the notes of about 15 μs. This is due to the fact that we are using a higher density of water and we have a 1/v absorber. But again, we are in the ball park.

Problem 7. Compute the number of absorptions per fission neutron.

Theoretically, since this is an infinite medium, all neutrons should eventually be removed from our system by absorption. However, if you put an energy cutoff into your code there is always a finite probability that a neutron will go below this cutoff. This is even more true when Hydrogen is the moderator. From the code results, the number of neutrons absorbed to that of fissioned was

$$\boxed{\frac{999129}{1000000} == 99.91\%.}$$

A Main Codes

A.1 Thermal Scattering Kernel Generation Code

```
% free gas thermal scattering kernel generation

% inputs
isoname = 'H_1';
savelib = 'no';
A = 1;
T = 300;
size = 10000;
kTfactor = [0.01,0.05,0.1,0.25,0.5,0.75,1,2,5,10,15,20,25,30,40,50,75,100,125,150,200,300];

% 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(size,length(kTfactor));
thermalcdf = zeros(size,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,size);

    % 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

% save output
if strcmp(savelib,'yes')
    kT = kTfactor;
    save(horzcat(isoname,'_therm'),'Erat','kT','thermalcdf');
end
```

A.2 Slowing Down by Particle

```
% Bryan Herman
% Slowing Down Code
% HW1 22.211
%
%
% clear close everything
```

```
clear
close all
clc
profile on
diary('results.txt');
% set input information
n_histories = 1000000;
Eo = 20.0;
A = 1;
seed = 5;
n_bins = 1000;
source = 'fission';

% print header
print_run_info();

% initalize objects
neut(1:n_histories) = particle_class();
tal = tally_class(n_bins,Eo);
pdf = pdf_class(seed);
mat = material_class(A);

% set materials (may put this in class later)
mat = mat.load_isotope('H_1',1);

% load thermal scattering libraries
pdf = pdf.load_thermal_lib('H_1');

% begin loop around histories
fprintf('\nBeginning Histories\n====================\n')
for j = 1:n_histories

    % sample source energy
    pdf = pdf.sample_source_energy(source);
    neut(j) = neut(j).set_energy(pdf.E);

    % perform scattering events until cutoff or absorbed
    while neut(j).alive == 1

        % calculate total macro xs
        mat = mat.compute_macro_totxs(neut(j).E);

        % bank neutron time
        neut(j) = neut(j).add_time(mat.totxs);

        % bank sampled energy (only 1 isotope for now)
        tal = tal.bank_tally(neut(j).E,mat.totxs);

        % sample reaction type
        pdf = pdf.sample_reaction(neut(j).E,mat.isotopes{1});

        % bank neutron collision information
        neut(j) = neut(j).collision(pdf.reaction);
```

```
        % check reaction type
        if strcmp(pdf.reaction,'capture')

            % kill neutron
            neut(j) = neut(j).kill;

        elseif strcmp(pdf.reaction,'scatter')

            % sample new energy
            pdf = pdf.sample_collision_energy(neut(j).E,mat.alpha);

            % set particle to that new energy
            neut(j) = neut(j).set_energy(pdf.E);

        else
            error('FATAL==> could not sample reaction type');
        end

        % check if neutron energy is too low (energy cutoff)
        if(neut(j).E < 1e-10)

            % kill particle
            neut(j) = neut(j).kill;

        end

        % check if neutron is dead
        if (neut(j).alive == 0)

            % reset tallies and bank flux
            tal = tal.reset;

        end

    end

    % display calculation progress
    if mod(j,1000) == 0
        fprintf('Histories: %d ...\n',j);

        % plot flux
        tal.plot_flux(1);
    end

end

% plot flux
tal.plot_flux(1);

profile viewer
diary off

% compute and print averages
compute_means(neut);
```

B Class Files

B.1 Thermal Scattering Library Class

```
classdef thermal_lib_class
    %THERMAL_LIB_CLASS defines the thermal scattering library object

    properties
        Erat
        cdf
        name
        kT
    end

    methods

        % constructor
        function obj = thermal_lib_class(name)

            % get name
            obj.name = name;

            % load xs data file expecting E and xs
            disp(horzcat('Loading thermal cdf file: ',name,'_therm'));
            load(horzcat('./pdfdata/',name,'_therm'));

            % set energy grid and xs
            obj.Erat = Erat;
            obj.cdf = thermalcdf;
            obj.kT = kT;

        end
    end

end
```

B.2 Particle Class

```
classdef particle_class
    %PARTICLE_CLASS defines a particle type for slowing down
    %   this class defines a particle along with its methods that will
    %   be used in the Monte Carlo slowing down code

    properties (SetAccess = private, GetAccess = public)
        E
```

```
    alive
    n_coll_therm
    n_coll
    n_abs
    time
    mass            % neutron mass in kg
end

methods

    % Constructor
    function obj=particle_class()
        obj.alive = 1;
        obj.n_coll_therm = 0;
        obj.n_coll = 0;
        obj.n_abs = 0;
        obj.time = 0;
        obj.mass = 1.674927351e-27;
    end

    % kill particle
    function obj=kill(obj)
        obj.alive = 0;
    end

    % sets particle energy
    function obj = set_energy(obj,E)
        obj.E = E;
    end

    % accumulate neutron lifetime
    function obj = add_time(obj,xs)

        % calculate velocity
        v = sqrt(2*obj.E*1.60217653e-13/obj.mass);

        % accumulate lifetime
        obj.time = obj.time + 1/(v*xs);

    end

    % increment the collision vars
    function obj = collision(obj,reaction)

        % bank collision info
        if obj.E > 1e-6
           obj.n_coll_therm = obj.n_coll_therm + 1;
        end
        obj.n_coll = obj.n_coll + 1;

        % bank absorption if absorbed
        if strcmp(reaction,'capture')
            obj.n_abs = obj.n_abs + 1;
        end
```

```
        end

        % increment the abs var
        function obj = increment_abs(obj)
           obj.n_abs = obj.n_abs + 1;
        end

    end

end
```

B.3 PDF Class

```
classdef pdf_class

    % PDF_CLASS contains information about the distribution function
    %   contains all of the random number sampling and pdfs needed by MC

    properties (SetAccess = private, GetAccess = public)

        rng    % random number generator object
        E      % new sampled energy
        egrid  % energy grid for chi pdf
        chicdf % cdf for chi
        n_thermal % number of thermal scattering libs
        thermal_list % list of thermal isotopes
        thermallibs % array of thermal scattering objs
        kT = 8.6173324e-5*300*10^-6; % boltzmann constant times temperature
        xs_ref = 7;
        E_ref = 0.025e-6;
        reaction

    end

    methods

        % constructor
        function obj = pdf_class(seed)

            % initialize random number generator with seed
            obj.rng = RandStream('mcg16807','Seed',seed);

            % create cdf
            obj = obj.watt_fission_cdf();

            % set thermal iso to zero
            obj.n_thermal = 0;

        end

        % sample energy
```

```
function obj = sample_collision_energy(obj,E,alpha)

    if E > 4e-6
        obj.E = E - E*(1-alpha)*obj.rng.rand;
    else

        % get a random number
        rn = obj.rng.rand;

        % create vector of energies
        Evec = zeros(1,size(obj.thermallibs{1}.cdf,2));

        % loop around a thermal kernels and get indices
        for i = 1:length(Evec)

            % get index
            idx = find(obj.thermallibs{1}.cdf(:,i).* ...
                 (obj.thermallibs{1}.cdf(:,i) > rn),1,'first');

            % get possible outgoing energy ratio
            Evec(i) = obj.thermallibs{1}.Erat(idx);

        end

        % interpolate energy ration to get correct outgoing energy
        obj.E = interp1(obj.thermallibs{1}.kT,Evec,E/obj.kT,...
            'linear','extrap')*E;

    end


end

% sample source energy
function obj = sample_source_energy(obj,opt)

    if strcmp(opt,'const')

        % assume fixed source at 2.0 MeV
        obj.E = 2.0;

    elseif strcmp(opt,'fission');

        % sample random number
        rn1 = obj.rng.rand;

        % find bin location
        idx = find(obj.chicdf.*(obj.chicdf > rn1),1,'first');
        obj.E = obj.egrid(idx);

    else
        error('FATAL==>Source cant be sampled.')
    end
```

```
        end

        % load thermal scattering library
        function obj = load_thermal_lib(obj,name)

            % increment number of isotopes
            obj.n_thermal = obj.n_thermal + 1;

            % call constructor of xsdata
            obj.thermallibs{obj.n_thermal} = thermal_lib_class(name);

            % append to list
            obj.thermal_list{obj.n_thermal} = name;

        end

        % sample reaction type
        function obj = sample_reaction(obj,E,iso)

            % compute absorption xs
            xs_a = sqrt(obj.E_ref/E)*obj.xs_ref;

            % get hydrogen scattering xs
            xs_s = interp1(iso.egrid,iso.xs,E,'linear','extrap');

            % compute probability of abs
            p = xs_a/(xs_a + xs_s);

            % sample random number
            rn = obj.rng.rand;

            % choose reaction type
            if rn < p
                obj.reaction = 'capture';
            else
                obj.reaction = 'scatter';
            end

        end

    end

    methods (Access = private)

        % construct cdf
        function obj = watt_fission_cdf(obj)

            % load cdf
            disp('Loading fission chi data...')
            load('./pdfdata/fission_chi','E','chicdf')

            % set properties
            obj.egrid = E;
            obj.chicdf = chicdf;
```

```
        end

    end

end
```

B.4 Material Class

```
classdef material_class
    %MATERIAL_CLASS Summary of this class goes here
    %   Detailed explanation goes here

    properties (SetAccess = private, GetAccess = public)
        A
        alpha
        n_isotopes
        isotopes
        iso_list
        totxs
        xs_ref = 7;
        E_ref = 0.025e-6;
    end

    methods

        % constructor
        function obj = material_class(A)

            % set vars
            obj.A = A;
            obj.alpha = ((A-1)/(A+1))^2;
            obj.n_isotopes = 0;
            obj.totxs = 0;

        end

        % import xsdata file
        function obj = load_isotope(obj,name,N)

            % increment number of isotopes
            obj.n_isotopes = obj.n_isotopes + 1;

            % call constructor of xsdata
            obj.isotopes{obj.n_isotopes} = cross_section_class(name,N);

            % append to list
            obj.iso_list{obj.n_isotopes} = name;

        end
```

```
        % get total xs
        function obj = compute_macro_totxs(obj,E)

            % reset totxs
            obj.totxs = 0;

            % begin loop around isotopes
            for i = 1:obj.n_isotopes

                % get macro total cross section
                xs_tot = interp1(obj.isotopes{i}.egrid,obj.isotopes{i}.xs,E,'linear','extra

                % get 1/v absorption cross section
                xs_a = sqrt(obj.E_ref/E)*obj.xs_ref;

                % compute total xs
                obj.totxs = obj.totxs + xs_tot + xs_a;

            end


        end

    end

end
```

B.5 Cross Section Class

```
classdef cross_section_class
    %CROSS_SECTION keeps track of ENDF xs data sets

    properties
        egrid
        xs
        name
        N
        totxs
    end

    methods

        % constructor
        function obj = cross_section_class(name,N)

            % get name
            obj.name = name;

            % load xs data file expecting E and xs
            disp(strcat('Loading xsdata file: ',name,'.m'));
            load(horzcat('./xsdata/',name));
```

```
            % set energy grid and xs
            obj.egrid = E;
            obj.xs = xs;

            % set number density
            obj.N = N;

            % save macroscopic total cross section
            obj.totxs = N*sum(obj.xs,2);

        end
    end

end
```

B.6 Tally Class

```
classdef tally_class
    %TALLY_CLASS defines the object for tallying MC quantites
    %   defines the object for tallying MC quantites

    properties(SetAccess = private,GetAccess = public)

        nbins   % number of energy grid bins
        egrid   % energy grid
        lgrid   % lethargy grid
        counts  % count
        lcounts % letharg bin counts
        eave    % array of average energy in egrid bins
        leave   % arrage of average energy in lethargy bins
        de      % energy spacing
        le      % lethargy spacing
        flux    % flux accumulation
        lflux   % flux in lethargy bins
        xs_ref = 7;
        E_ref = 0.025e-6;

    end

    methods

        % constructor
        function obj = tally_class(nbins,Eo)

            % initalize values
            obj.nbins = nbins;
            obj.egrid = linspace(0,Eo,nbins+1);
            obj.eave = linspace((Eo/(nbins*2)),Eo-(Eo/(nbins*2)),nbins);
            obj.counts = zeros(1,nbins);
            obj.flux = zeros(1,nbins);
```

```
        obj.de = obj.egrid(2) - obj.egrid(1);

        obj.lgrid = logspace(-10,log10(Eo),nbins+1);
        obj.leave = 0.5*(obj.lgrid(1:nbins) + obj.lgrid(2:nbins+1));
        obj.lcounts = zeros(1,nbins);
        obj.lflux = zeros(1,nbins);
        obj.le = log(max(obj.lgrid)/obj.lgrid(1)) - ...
                 log(max(obj.lgrid)/obj.lgrid(2));

    end

    % bank tally
    function obj = bank_tally(obj,E,totxs)

        % find out bin index
        idx = find(obj.egrid.*(obj.egrid > E),1,'first')-1;

        if idx > length(obj.egrid) || idx == 0
            error('FATAL ==> index out of bounds');
        end

        % bank a count
        obj.counts(idx) = obj.counts(idx) + 1/totxs;

        % find out lethargy bin index
        idx = find(obj.lgrid.*(obj.lgrid >= E),1,'first')-1;

        % bank a count (disregard below the cutoff energy)
        if (idx > 0)
            obj.lcounts(idx) = obj.lcounts(idx) + 1/totxs;
        end

    end

    % plot tally
    function plot_tally(obj,n)

        % get random rgb code
        r = rand(1);
        g = rand(1);
        b = rand(1);

        figure(n);
        hold on;
        plot(obj.eave,obj.counts,'Color',[r,g,b]);

    end

    % plot flux
    function plot_flux(obj,n)

        %figure(n)
        %loglog(obj.eave,obj.flux);
```

```
            figure(n)
            loglog(obj.leave,obj.lflux);
            drawnow;
        end

        % clear tally
        function obj = reset(obj)

            % bank in flux first
            obj.flux = obj.flux + obj.counts;
            obj.lflux = obj.lflux + obj.lcounts;

            % now reset
            obj.counts(:) = 0;
            obj.lcounts(:) = 0;

        end

    end

    methods(Access = private)

        % determine the index on the energy grid
        function idx = get_energy_idx(obj,E)

            idx = ceil(E/obj.de);

        end

        % determine the index on the lethargy grid
        function idx = get_lethargy_idx(obj,E)

            % this is complex since we are putting on a log energy grid
            l = log(max(obj.lgrid)/E);
            idx = length(obj.lgrid) - floor(l/obj.le) - 1;
            if idx <= 0
                idx = 1;
            end

        end

    end

end
```