11.07.2019

Uncertainty calculations with OpenMOC for lattice reactor physics

Table of contents

1. Introduction

The purpose of this project is to use the open source code “OpenMOC” [1] for uncertainty and sensitivity analyses in the generation of macroscopic cross sections for light water reactors (LWRs).

OpenMOC belongs to the class of spectral codes used in reactor core design. Their main application is to generate few energy-group, spatially homogenized cross sections (also called homogenized macroscopic cross sections) from microscopic cross section data. These macroscopic cross sections can then be used in full-core reactor simulators to calculate the power distribution and its burnup dependence. These simulators typically apply nodal neutron transport solvers in which the reactor core is approximated as a set of a small number of coupled nodes. Each of these nodes is assumed to have a homogenous material composition and a few-energy group dependence as described by its macroscopic cross sections. Different nodes can have different material properties because of different burnups (i.e. nuclide composition) or because they belong to different fuel assembly types (i.e. UOX or MOX fuel).

In principle a full-core reactor power distribution can also be calculated directly from microscopic cross section data with Monte Carlo codes such as MCNP [2] or OpenMC [3]. In 2019 this still requires high performance computer clusters and days of computation time while the method of combining results from a spectral code together with a nodal neutron transport solver takes only a few minutes of calculations and it was already successfully used in the licensing of LWRs in the 1970s []. One drawback of the latter method is the difficulty to quantify the impact of the microscopic cross section uncertainty on the core power distribution.

A focus of ongoing research is to augment best-estimate microscopic cross sections with covariance data to enable uncertainty and sensitivity analyses of core power distributions [4,5,6]. One of the newest approach for this purpose is the “Total Monte Carlo” method (TMC) [7,8]: given the covariance data an ensemble of microscopic cross sections is generated (i.e. sampled according to the covariance information), then an ensemble of macroscopic cross sections is derived and finally a set of resulting reactor core power distributions is determined. The TMC method could directly and naturally be used together with classical Monte Carlo codes like MCNP – but the computational burden would still be infeasible. Hence the research focus currently is on enabling spectral codes and nodal reactor simulators for this purpose.

In this project the generation of an ensemble of homogenized macroscopic cross sections without a reactor core simulation is pursued. The primary reason for using OpenMOC is its

open-source status while all other spectral codes known to the author are proprietary. This enables open communication and transparency of results and source code modification. Another advantage of OpenMOC is its ability to utilize GPUs [9]. Hence its computations are very fast compared to all other spectral codes known by the author and is therefore particularly suited to generate large ensembles of cross sections.

The structure of the text is as follows: the introduction continues with an explanation of the basic theory of neutron transport in LWRs, the description of the microscopic cross section data available for LWR applications, the structure and method of the OpenMOC code and of the microscopic data reader NJOY [10,11].

Chapter 2 describes the OpenMOC code in detail together with the coded Python scripts to generate and analyze ensembles of macroscopic cross sections, given an ensemble of microscopic cross sections.

In chapter 3 the methods of the microscopic cross section reader NJOY are explained together with the coded Python scripts to generate an ensemble of microscopic cross sections, given the covariance data.

Finally, chapter 4 contains the results of the TMC method done with OpenMOC and a comparison with best estimate results from a proprietary spectral code.

Chapter 5 gives recommendations for improvements in future work and an outlook how to use the results in a reactor core simulator.

1.1. Boltzmann equation for neutron transport

The Boltzmann equation of neutron transport [12] determines the neutron flux in a reactor core and in the most general case is formulated as follows (for an explanation of the different terms see for example [13]):

(1) $\Sigma_t(\vec{x},E,t)\varphi\left(\vec{x},\hat{\Omega},E,t\right)+\hat{\Omega}\cdot\nabla\varphi+\frac{1}{v(E)}\cdot\frac{\partial\varphi}{\partial t}=\int_0^{4\pi}d\hat{\Omega}'\int_0^{\infty}dE'\,\Sigma_s\left(\vec{x},E'\rightarrow E,\hat{\Omega}'\rightarrow\hat{\Omega},t\right)\varphi+$
$+\frac{\chi(\vec{x},E)}{4\pi}\int_0^{\infty}dE'v(\vec{x},E',t)\Sigma_f(\vec{x},E',t)\varphi$

In (1) the angular flux φ is a function of location $\vec{x}$, direction $\hat{\Omega}$, energy E and time t. It is defined as the product of the angular neutron density n and the neutron speed $\mathbb{v}$:

(2) $\varphi\left(\vec{x},\hat{\Omega},E,t\right)=\mathbb{v}(E)\cdot n\left(\vec{x},\hat{\Omega},E,t\right)$

The macroscopic cross sections Σ are the product of the number density ρ of the material and its microscopic cross section σ:

(3) $\Sigma = \rho(\vec{x},t) \cdot \sigma(\vec{x},E,t)$

The energy spectrum of the fission neutrons is $\chi(\vec{x},E)$ and the neutron numbers per fission event is $\upsilon(\vec{x},E,t)$.

In (1) the total cross section Σ_t, the scattering cross section Σ_s and the fission cross section Σ_f are occurring. These macroscopic cross sections are a function of energy and space (and sometimes of time, too) and must be spatially averaged and homogenized and reduced to a few-group energy dependence for subsequent use in a nodal solver for a reactor core simulation.

If a material region consists of a homogenous mixture of materials, the cross section definition for type Σ_x (x = fission/scatter/total) is:

(4) $\Sigma_x = \sum_k \rho_k(\vec{x}) \cdot \sigma_{x,k}(\vec{x},E)$

The sum is over all nuclide types k within the homogenous mixture.

If the solution φ_{true} to (1) is known, a few-energy group structure can be formulated (stationary solution and factorization assumption $\varphi_{true} \approx f_1(\vec{x},E)f_2(\vec{x},\hat{\Omega})$):

(5) $\Sigma_{t,g}(\vec{x})\varphi_g(\vec{x},\hat{\Omega}) + \hat{\Omega}\cdot\nabla\varphi_g = \int_0^{4\pi} d\hat{\Omega}' \sum_{g'} \Sigma_{s,g'\to g}(\vec{x},\hat{\Omega}'\to\hat{\Omega})\varphi_{g'} + \frac{\chi_g}{4\pi}\sum_{g'} \upsilon_{g'}\Sigma_{f,g'}(\vec{x})\,\varphi_{g'}$

$$\varphi_g = \int_{E_g}^{E_g+\Delta} dE' \varphi_{true}(\vec{x},\hat{\Omega},E')$$

$$\Sigma_{x,g}(\vec{x}) \cdot \varphi_g(\vec{x},\hat{\Omega}) = \int_{E_g}^{E_g+\Delta} dE' \Sigma_x(\vec{x},E')\varphi_{true}(\vec{x},\hat{\Omega},E')$$

The sum g is over G groups. In LWR cases typically G=2 or 4.

In practice the following procedure has been applied for LWR applications: a high-resolution energy solution φ_{cell} is determined for a unit cell like a fuel rod plus its surrounding subchannel and then the location and angular dependence is averaged out. This result $\varphi_{true} \approx \varphi_{cell}$ is then used in the integrals in (5) and in practice it has been shown that this approach is often sufficiently accurate. Programs like NJOY contain reference functions for φ_{cell} for this purpose.

The next step is to reduce the dependency of (5) on the neutron direction $\hat{\Omega}$. Since the fission reaction is to a very good approximation isotropic and the scattering cross section depends only on the change in direction, (5) can be further simplified. The angular flux can be expressed in terms of spherical harmonics and the scattering cross section as a Legendre polynomial:

(6) $\varphi_g(\vec{x},\hat{\Omega}) = \sum_{l=0}^{L} \frac{2l+1}{4\pi} \sum_{m=-l}^{+l} \phi_{g,l,m}(\vec{x}) Y_{l,m}(\hat{\Omega})$;

$$\Sigma_{s,g'\to g}(\vec{x},\hat{\Omega}'\to\hat{\Omega}) = \Sigma_{s,g'\to g}(\vec{x},\hat{\Omega}'\cdot\hat{\Omega}) = \sum_{m=-l}^{+l} \Sigma_{s,g'\to g,l}(\vec{x}) P_l(\hat{\Omega}'\cdot\hat{\Omega})$$

The direction independent neutron flux $\phi_g = \phi_{g,0,0}$ and the currents can be calculated as (not writing down the energy dependence explicitly) in the P1 approximation:

(7) $\phi(\vec{x}) = \phi_{0,0}(\vec{x}) = \int d\hat{\Omega}\, \varphi(\vec{x},\hat{\Omega})$;

$$J_x = \int d\hat{\Omega}\, (\hat{n}_x\cdot\hat{\Omega})\varphi(\vec{x},\hat{\Omega}) = \frac{1}{\sqrt{2}}(\phi_{1,-1} - \phi_{1,1})$$

$$J_y = \int d\hat{\Omega}\, (\hat{n}_y\cdot\hat{\Omega})\varphi(\vec{x},\hat{\Omega}) = \frac{1}{\sqrt{2}}(\phi_{1,-1} + \phi_{1,1})$$

$$J_z = \int d\hat{\Omega}\, (\hat{n}_z\cdot\hat{\Omega})\varphi(\vec{x},\hat{\Omega}) = \phi_{1,0}$$

Then the angular flux and the current becomes (Fick's law):

(8) $\varphi_g(\vec{x},\hat{\Omega}) = \frac{1}{4\pi}\phi_g(\vec{x}) + \frac{3}{4\pi}\hat{\Omega}\cdot J_g(\vec{x})$

$$\vec{J}_g(\vec{x}) = -\frac{1}{3\left(\Sigma_{t,g}(\vec{x}) - \bar{\mu}_o \Sigma_{s,g}(\vec{x})\right)} \nabla\phi_g(\vec{x}) = -D_g(\vec{x})\cdot\nabla\phi_g(\vec{x})$$

The scalar flux (averaged over energy, too) ϕ is often used in a product with the macroscopic cross sections to determine the reaction rates R:

(9) $R(\vec{x}) = \Sigma_x(\vec{x})\phi(\vec{x})$

As will be shown later one of the most important principles of cross section homogenization is to preserve reaction rates like (9) per node or per fuel assembly.

The above approximations finally lead to the diffusion approximation of the Boltzmann equation:

(9) $\Sigma_{t,g}(\vec{x})\phi_g(\vec{x}) - \nabla\cdot D_g(\vec{x})\nabla\phi_g(\vec{x}) = \sum_{g'} \Sigma_{s,g'\to g,0}(\vec{x})\phi_{g'} + \chi(E_g) \sum_{g'} \nu_{g'}\Sigma_{f,g'}(\vec{x})\,\phi_{g'}$

Equation (9) has been the basis for all LWR reactor design calculations in the 1970s and later. There are many instances where it is a very poor approximation: for example, near strongly absorbing regions like in the vicinity of control rods or near the core reflector. Also, in a heterogenous reactor core in which high and low burnup fuel assemblies stand side-by-side or in which UOX and MOX fuel assemblies share a common boundary, equation (9) is also often not a good approximation. However, modern nodal reactor simulation codes have been augmented with modifications to eq. (0) which allow it to be sufficiently accurate in practice (e.g. [14,15]).

For most LWR applications the time dependence in (1) is dropped. This is strictly true if the neutron field is determined only for steady state operation. The time dependence enters on a relatively long timescale of many hours and days in the form of burnup: as the energy generation by fission progresses, fission products are accumulating in the fuel which in turn change the number densities of the cross sections in (3). This time dependence is typically treated in a quasi-static manner: (1) is solved without time dependence, a burnup step is calculated with constant neutron field and then (1) is updated with new cross section data. The equations describing the time dependence of the nuclide vector are called the Bateman equations [16] describing the generation and destruction of nuclides:

(10) $\frac{dN_k}{dt} = \sum_{j \neq k}\left(l_{kj}\lambda_j + f_{kj}\sigma_j\phi\right)N_j - (\lambda_k + \sigma_k\phi)N_k + S_k$

In a reactor simulator every relevant core region (e.g. a fuel assembly node) is approximated as an infinite, homogenous medium in which fission products are generated and decaying and higher actinides are bred through neutron capture, and alpha and beta decay channels. The number l_{kj} describes the fractional yield of nuclide k from decay of nuclide j, the number f_{kj}describes the fractional yield of nuclide k from neutron induced removal of nuclide j in reaction with cross section σ_j, the nuclide N_k decays with constant λ_k and the fission yield source term is described by S_k.

Another reason why time dependence enters (1) is due to the delayed neutrons which are emitted by neutron rich fission products. They are the basic reason why the chain reaction in commercial reactors can be controlled by practical means. In this case (1) is augmented by the following set of equations:

(11) $\Sigma_t(\vec{x},E,t)\varphi\left(\vec{x},\hat{\Omega},E,t\right) + \hat{\Omega}\cdot\nabla\varphi + \frac{1}{v(E)}\cdot\frac{\partial\varphi}{\partial t} = \int_0^{4\pi} d\hat{\Omega}' \int_0^{\infty} dE'\,\Sigma_s\left(\vec{x},E'\to E,\hat{\Omega}'\to\hat{\Omega},t\right)\varphi +$

$+(1-\beta)\frac{\chi_p(E)}{4\pi}\int_0^{\infty} dE' v(\vec{x},E',t)\Sigma_f(\vec{x},E,t)\varphi + \frac{\chi_d(E)}{4\pi}\sum_k \lambda_k C_k(\vec{x},t)$

$\frac{dC_k}{dt} = \beta_k \int_0^{\infty} dE' v(\vec{x},E',t)\Sigma_f(\vec{x},E,t)\varphi - \lambda_k C_k$

The fraction of delayed neutrons is β and the energy spectrum of the delayed and prompt neutrons has been named χ_d and χ_p, respectively.

Typically, a k = 6 or 8 group structure of delayed neutron precursors is considered, i.e. all relevant fission products are grouped according to their decay constants λ_k before emitting a neutron. It is $\sum_k \beta_k = \beta$.

From (11) the classical point reactor kinetics equations can be derived. Details of the derivation can be found in [17], for example. For the angular velocity the following Ansatz is made: $\varphi\left(\vec{x},\hat{\Omega},E,t\right) = N(t)\cdot\Psi\left(\vec{x},\hat{\Omega},E,t\right)$. The time dependence of flux Φ is split into a fast compo-

nent $N(t)$ and a slow component $\Psi(\mathrm{t})$. The corresponding adjoint solutions are $\Psi^{\dagger}$ and $\Phi^{\dagger}$. Let Ψ fulfill the following eigenvalue equation (neglecting the delayed neutron contributions):

(12) $\Sigma_t(\vec{x},E,t)\Psi\left(\vec{x},\widehat{\Omega},E,t\right)+\widehat{\Omega}\cdot\nabla\Psi=\int_0^{4\pi}d\widehat{\Omega}'\int_0^{\infty}dE'\,\Sigma_s\left(\vec{x},E'\to E,\widehat{\Omega}'\to\widehat{\Omega},t\right)\Psi+$
$+(1-\rho(t)\,)\frac{\chi_p(E)}{4\pi}\int_0^{\infty}dE'\upsilon(\vec{x},E',t)\Sigma_f(\vec{x},E,t)\Psi$

The time dependence of Ψ enters through the time dependence of the cross sections and not because of any time derivative in (12). The quantity $\rho(t)$ plays the role of fundamental eigenvalue and must be determined so that there exists a positive solution Ψ. The main assumption for the following equations is that the time dependent cross sections vary on the same slow timescale on which the neutron precursors decay and that the system is near criticality at all times. The following definitions are made:

(13) $\theta(\mathrm{t})=\frac{\int\ dV\int_{4\pi}d\widehat{\Omega}\int\ dE\Psi^{\dagger}\frac{1}{v}\cdot\frac{d\Psi}{dt}}{\int\ dV\int_{4\pi}d\widehat{\Omega}\int\ dE\Psi^{\dagger}\frac{1}{v}\cdot\Psi}$

$\frac{1}{\Lambda(\mathrm{t})}=\frac{\int\ dV\left(\int\ dE\Phi^{\dagger}\chi_p\right)\left(\int\ dE\upsilon\Sigma_f\Phi\right)}{4\pi\int\ dV\int_{4\pi}d\widehat{\Omega}\int\ dE\Psi^{\dagger}\frac{1}{v}\cdot\Psi}$

$\beta(\mathrm{t})=\frac{\int\ dV\left(\int\ dE\Phi^{\dagger}\chi_p\right)\left(\int\ dE\upsilon\beta\Sigma_f\Phi\right)}{\int\ dV\left(\int\ dE\Phi^{\dagger}\chi_p\right)\left(\int\ dE\upsilon\Sigma_f\Phi\right)}$

$c_j(\mathrm{t})=c_j(0)e^{-\lambda_j t}+\frac{\int\ dV\left(\int\ dE\Phi^{\dagger}\chi_p\right)\left[\int_0^t dt'e^{-\lambda_j(t-t')}\left(\int\ dE\upsilon\beta_j\Sigma_f\Phi\right)N(t')\right]}{4\pi\int\ dV\int_{4\pi}d\widehat{\Omega}\int\ dE\Psi^{\dagger}\frac{1}{v}\Psi}$

Then it follows:

(14) $\frac{dN}{dt}+\theta(\mathrm{t})N(t)=\frac{\rho(t)-\beta(\mathrm{t})}{\Lambda(\mathrm{t})}\cdot N(t)+\sum_j\ \lambda_j c_j(\mathrm{t})$

And if the time dependence of $\Phi^{\dagger}$ is small:

(15) $c_j(\mathrm{t})=c_j(0)e^{-\lambda_j t}+\int_0^t dt'e^{-\lambda_j(t-t')}\frac{\beta_j(\mathrm{t}')}{\Lambda(\mathrm{t}')}N(t')$

If $\theta(\mathrm{t})$ is small, then equation (14) and (15) become the classical point reactor kinetic equations.

1.2. Microscopic cross sections

Neutrons can either be scattered or absorbed. Scattering can be elastic or inelastic, and in this case they survive the interaction. Absorption can mean capture and creation of a daughter nuclide which subsequently may experience beta decay. Also, photons are often emitted during de-excitation of the daughter's internal nuclide states. Absorption can also mean re-emission of a secondary particle or of a set of particles like another neutron, proton or alpha particle together with photons. Finally, absorption can also lead to fission and other reactions.

The microscopic scattering cross sections are proportional to the probability that a given reaction channel is occurring. Therefore $\sigma(E, E', \hat{\Omega}, \hat{\Omega}')$ is proportional to the probability that a neutron with energy E and direction $\hat{\Omega}$ is scattered into energy E' and direction $\hat{\Omega}'$. Absorption reactions like (n,γ), (n,α), (n,2n') or (n, fission) depend only on the incoming neutron energy and cross sections have a $\sigma(E)$-like functional dependence. If the secondary particle field is of interest, too, probabilities for secondary particle energy and direction must be given as additional cross section dependences.

Microscopic cross section data for neutron interactions are available in several libraries, for example ENDF/B-VII.1 [18] or JEFF-3.3 [19]. The cross sections are stored in a standardized format, called the ENDF (evaluated nuclear data file) format libraries [20]. For practical use the data must be preprocessed and the most common code for this purpose is NJOY [21]. To inspect the data tools like JANIS (https://www.oecd-nea.org/janis/) or Sigma (https://www.nndc.bnl.gov/sigma/) can be used.

The formats were upgraded with each version to handle new features, for example, the extension from the original upper limit of 15 MeV to 20 MeV, the addition of photon production information, the introduction of new resonance formats, or the addition of charged-particle data. Control over the ENDF formats has been retained by the US Cross Section Evaluation Working Group and the format specifications are published through the National Nuclear Data center (https://www.nndc.bnl.gov/).

1.3. Cross section homogenization

Koepke's generalized equivalence theory [22,23] today has become one of the standard methods to improve the accuracy of (9) and is the basis of many existing nodal solvers for LWR core simulators. The following assumptions are made:

- all group reaction rates are preserved per node

  (16) $\int_V dV\ \Sigma_{k,g}\phi_g = \int_V dV\ \tilde{\Sigma}_{k,g}\tilde{\phi}_g$

  here ϕ_g is the exact solution of (9) and $\tilde{\phi}_g$ is the homogenized flux. The exact cross section is $\Sigma_{k,g}$ and the homogenized one is $\tilde{\Sigma}_{k,g}$. The latter per definition has no spatial dependence within a node and hence can be taken out of the above integral:

  (17) $\tilde{\Sigma}_{k,g} = \int_V dV\ \Sigma_{k,g}\phi_g \Big/ \int_V dV\ \tilde{\phi}_g$

- all group surface currents are preserved
  in diffusion approximation the condition for surface j of volume i () is:

  (18) $\int_{S_{i,j}} dS\ D_g \nabla\tilde{\phi}_g = -\int_{S_{i,j}} dS\ \vec{J}_g$

  Here $\vec{J}_g$ is the exact surface current

- keff of the total system is preserved.

The above conditions lead to the challenge that the homogenized diffusion coefficients depend of the surface on which they have been defined. The main idea of Koepke was to add additional free parameters in the form of 'discontinuity' factors. With these factors the homogenized flux no longer must be continuous across node boundaries but makes discrete jumps while the current is still preserved also in diffusion approximation. At a node boundary x_i the factors are as follows:

(19) $f_{g,i}^{x+} = \frac{\phi_g(x_i)}{\tilde{\phi}_g(x_i,+)}$ (one the left side); $f_{g,i+1}^{x-} = \frac{\phi_g(x_i)}{\tilde{\phi}_g(x_i,-)}$ (on the right side)

The homogenized fluxes $\tilde{\phi}_g(x_i,+)$ and $\tilde{\phi}_g(x_i,-)$ at the left and right hand side of a node boundary are therefore related according to

(20) $f_{g,i}^{x+}\tilde{\phi}_g(x_i,+) = f_{g,i+1}^{x-}\tilde{\phi}_g(x_i,-)$.

For many LWR applications the discontinuity factors can to very good approximation be determined by the following equation:

(21) $f_g^{S_i} = \frac{1}{S_i}\int_{S_i} dS\phi_g \Big/ \frac{1}{V}\int_V dV\phi_g$

Most spectral code solve (21) with a high resolution approximation for ϕ_g. Subsequently reactor core simulators can use the $f_g^{S_i}$ to determine the global reactor solution for $\tilde{\phi}_g$.

In the nodal solution method the neutron flux $\tilde{\phi}_g$ is integrated over the surface perpendicular to the principal direction of each node i:

(23) $\widetilde{\Phi}_{g,x}(x_i) = \int_{S_{i,x\perp}} dS\, \tilde{\phi}_g(x_i, y, z)$

$$\widetilde{\Phi}_{g,y}(y_i) = \int_{S_{i,y\perp}} dS\, \tilde{\phi}_g(x, y_i, z)$$

$$\widetilde{\Phi}_{g,z}(z_i) = \int_{S_{i,z\perp}} dS\, \tilde{\phi}_g(x, y, z_i)$$

By integrating (9) over these surfaces, too, three equations for each $\widetilde{\Phi}_g$ in (23) can be derived. All material properties within a node are constant, assuming homogenized quantities:

(24) $S_{i,x\perp}\tilde{\Sigma}_{t,g}(i)\widetilde{\Phi}_{g,x}(x) - D_g(i)\left(S_{i,x\perp}\partial_x^2\widetilde{\Phi}_{g,x} + \int_{S_{i,x\perp}} dS\left(\partial_y^2\widetilde{\Phi}_g + \partial_y^2\widetilde{\Phi}_g\right)\right) =$
$= S_{i,x\perp}\sum_{g'}\left(\Sigma_{s,g'\to g}(i) + \chi_g \nu_{g'}\Sigma_{f,g'}(i)\right)\widetilde{\Phi}_{g'}$

A similar equation holds for $\widetilde{\Phi}_{g,y}$ and $\widetilde{\Phi}_{g,z}$. The third term on the LHS of (24) is the net transverse leakage $L_g^x(x_i)$ for the node i with regard to the principal direction x. This leads to three second order differential equations with constant coefficients and a source term; e.g. in the x-direction:

(25) $-D_g(i)\partial_x^2\widetilde{\Phi}_{g,x} + \tilde{\Sigma}_{t,g}(i)\widetilde{\Phi}_{g,x} - \sum_{g'}\left(\Sigma_{s,g'\to g}(i) + \chi_g \nu_{g'}\Sigma_{f,g'}(i)\right)\widetilde{\Phi}_{g'} = -L_g^x(x)$

The first boundary condition is that at each surface the net current in the respective direction is preserved. In the diffusion approximation this is calculated by the gradient of $\widetilde{\Phi}_g$:

(26) $-D_g(i)\partial_x\,\widetilde{\Phi}_{g,x} = \int_{S_{i,x\perp}} dS\, J_{x,g}(x, y, z)$

The second boundary condition is that the flux $\widetilde{\Phi}_g$ on the left and right side of a node boundary is connected via the discontinuity factors (ADFs):

(27) $f_{g,i}^{x+}\widetilde{\Phi}_{g,x+} = f_{g,i+1}^{x-}\widetilde{\Phi}_{g,x-}$

In practice a solution to (25) is found by starting with an estimate of the leakage terms L and iteratively determine a self-consistent solution. The ADFs are usually provided together with the homogenized cross section estimates from a spectral code like OpenMOC.

1.4. Methods of characteristics

The method of characteristics (MOC) is one of several techniques for solving partial differential equations. MOC is one of the most widely used method employed by commercial spectral codes [Smith]. In spectral codes MOC is typically used to solve the Boltzmann equation in 2D by discretizing both polar and azimuthal angles and integrating the multi-group form of the equation for a particular azimuthal and polar angle quadrature [24].

In order to formulate the MOC method eq. (1) is rewritten as follows (stationary case):

(28) $\Sigma_t(\vec{x},E)\varphi(\vec{x},\hat{\Omega},E)+\hat{\Omega}\cdot\nabla\varphi=Q(\vec{x},\hat{\Omega},E)$

The source term Q contains the fission and scattering source term. The location vector $\vec{x}$ is parametrized:

(29) $\vec{x}=\vec{x}(s,\hat{\Omega})=\vec{x}_0+s\hat{\Omega}$

Eq. (x) can now be rewritten (characteristic form):

(30) $\Sigma_t(\vec{x}_0+s\hat{\Omega},E)\varphi(\vec{x}_0+s\hat{\Omega},\hat{\Omega},E)+\frac{\mathrm{d}}{ds}\varphi(\vec{x}_0+s\hat{\Omega},\hat{\Omega},E)=Q(\vec{x}_0+s\hat{\Omega},\hat{\Omega},E)$

For a given ray the solution of (30) can be written as:

(31) $\varphi(s,\hat{\Omega},E)=\varphi(0,\hat{\Omega},E)e^{-\int_0^s ds\prime\Sigma_t(s\prime,E)}+\int_0^s ds''Q(s'',\hat{\Omega},E)\cdot e^{-\int_{s\prime}^s ds\prime\Sigma_t(s\prime,E)}$

Again, the above equation is discretized with respect to energy:

(32) $\varphi_g(s,\hat{\Omega})=\varphi_g(0,\hat{\Omega})e^{-\int_0^s ds\prime\Sigma_{t,g}(s\prime)}+\int_0^s ds''Q_g(s'',\hat{\Omega})\cdot e^{-\int_{s\prime}^s ds\prime\Sigma_{t,g}(s\prime)}$

The multi-group form of the source Q then takes the form:

(33) $Q_g(s,\hat{\Omega})=\int_0^{4\pi}d\hat{\Omega}'\sum_{g'=1}^{G}\Sigma_{s,g'\to g}(\vec{x},\hat{\Omega}'\to\hat{\Omega})\varphi_{g'}(s,\hat{\Omega}')+$
$+\ \frac{\chi_g}{4\pi}\sum_{g'=1}^{G}\nu_{g'}\Sigma_{f,g'}(s)\int_0^{4\pi}d\hat{\Omega}'\,\varphi_{g'}(s,\hat{\Omega}')$

Next a discretization of ordinates is done with weighting factors ω_m:

(34) $\varphi_g(s)=\int_0^{4\pi}d\hat{\Omega}'\varphi_g(s,\hat{\Omega}')\approx\sum_{m=1}^{M}\omega_m\varphi_g(s,\hat{\Omega}_m)=\sum_{m=1}^{M}\omega_m\varphi_{g,m}(s)$

Another common approximation for MOC is to assume that the source Q is constant across discrete spatial cells termed *flat source regions* (FSRs). This implies that the source does not vary along a characteristic entering a FSR. Also, it is assumed that the material properties are constant across each FSR. In practice a finite set of characteristics, also known as tracks, are used to solve (32). In practice MOC codes usually have three iterations loops: a ray-tracing loop, an energy loop and a fission loop. During the ray-tracing loop each track is broken into segments as it crosses different FSRs. The angular flux approximation from a previous loop or from the first guess at the problem domain boundary is propagated along each FSR until the track exists boundary the geometry. From this information the scalar flux can be approximated. In the energy loop it is propagated from the highest to the lowest energy group. This result in turn then is used to estimate the fission source and the procedure restarts until a given level of accuracy is reached.

1.5. Uncertainty and sensitivity analysis

The cross section evaluators will form at least a subjective opinion about the joint probability distribution of the data they have examined and they will express it as a joint probability distribution p:

(35) $p(\sigma_{r1}, \sigma_{r2}, \dots)\, d\sigma_{r1} d\sigma_{r2} \dots$

It means that the probability of the cross section for reaction r1, r2,… to be in the interval σ_{r1} to $\sigma_{r1} + d\sigma_{r1}$,… is p. In the early versions of the ENDF format only the first moments of this probability distribution existed. Beginning with ENDF/B-IV and later the second moments of the data probability distributions have been included in many of the files. These second moments (covariances) contain information on the uncertainty of individual data as well as correlations that may exist.

The determination of a cross section σ_{r1} is subject to random influences. The true value then becomes the expectation value of a random variable:

(36) $\sigma_{r,o} = \mathrm{E}(\sigma_r)$

The second moment of the joint probability distribution between r1, r2 is the covariance:

(37) $\mathrm{cov}(\sigma_{r1}, \sigma_{r2}) = \mathrm{E}((\sigma_{r1} - \sigma_{r1,0})(\sigma_{r2} - \sigma_{r2,0}))$

The covariance is a measure of the degree to which both cross sections are both affected by the same source of errors. The covariance of r1 with itself is the variance:

(38) $\mathrm{var}(\sigma_{r1}) = \mathrm{cov}(\sigma_{r1}, \sigma_{r1}) = \mathrm{E}((\sigma_{r1} - \sigma_{r1,0})^2)$

The covariance data can be extracted with the help of the ERRORR module of NJOY from an ENDF data tape. It is necessary to begin the procedure by converting the energy-dependent covariance information into multigroup form. These multigroup data are obtained from the GROUPR processing of NJOY.

Given a covariance matrix it is necessary for the Total Monte Carlo Method to generate an ensemble of cross sections. Assume that the number of energy groups is N_g and that the number of samples in the ensemble is N_s. First, a matrix $N_g \times N_s$ (with name R) with normally distributed random values is created. Then the covariance matrix Σ has the dimension $N_g \times N_g$ and it is positive definite and can be decomposed with the Cholesky decomposition [25] as follows:

(39) $\Sigma = \mathrm{C} \cdot C^T$

In (39) the matrix C is lower triangular. The matrix product of R with C yields the desired $N_g \times N_s$ matrix D in which each column vector is a set of N_g random variables distributed according to the covariance matrix Σ.

(40) $\mathrm{D} = \mathrm{C} \cdot \mathrm{R}$

This can be seen as follows:

(41) $\mathrm{E}(\mathrm{D}) = \mathrm{E}(\mathrm{C} \cdot \mathrm{R}) = \mathrm{C} \cdot \mathrm{E}(\mathrm{R}) = C \cdot 0 = 0$

$$\mathrm{E}(D \cdot D^T) = \mathrm{E}(\mathrm{C} \cdot \mathrm{R} \cdot (\mathrm{C} \cdot \mathrm{R})^T) = \mathrm{E}(\mathrm{C} \cdot \mathrm{R} \cdot \mathrm{R}^T \cdot \mathrm{C}^T) = \mathrm{C} \cdot \mathrm{E}(\mathrm{R} \cdot \mathrm{R}^T) \cdot \mathrm{C}^T = \mathrm{C} \cdot \mathrm{C}^T = \Sigma$$

1.6. The OpenMOC code

OpenMOC is a Method of Characteristics (MOC) neutral particle transport code for reactor physics criticality calculations (https://mit-crpg.github.io/OpenMOC/). It is capable of simulating 2D assembly and full-core reactor models based on constructive solid geometry with second-order surfaces. High-performance parallel solvers for multi-core CPUs and GPUs are actively pursued as part of the OpenMOC project.

The OpenMOC source code is hosted on GitHub (https://github.com/mit-crpg/OpenMOC) and is provided to users as a Python API. The back-end of the code is written in C/C++ and CUDA [26].

A basic model consists of:

- **Geometry**: a description of the geometry split into regions of homogeneous materials
- **Materials**: a description of the nuclear cross-sections for each material
- **Parameters**: various parameters for the numerical algorithm used in the simulation.

OpenMOC uses constructive solid geometry (CSG) to represent complex reactor models.The constructive solid geometry formulation is the method of choice for many advanced reactor modeling software packages. CSG allows complex spatial models to be built using Boolean operations - such as intersections and unions - of simple surfaces and building blocks termed *primitives*. This is well suited for commercial LWRs whose cores are built out of a rectangular lattice of fuel assemblies, each of which is a rectangular lattice of fuel pins.

The constructive solid geometry formulation in OpenMOC is predicated upon the use of several key objects which allow one to construct a spatial model from simple primitives in a hier-

archical fashion. OpenMOC is presently only capable of describing the 2D xy-plane. A typical hierarchy for the way in which surfaces, universes, cells and lattices are constructed to represent a reactor model in OpenMOC is illustrated in figure x.

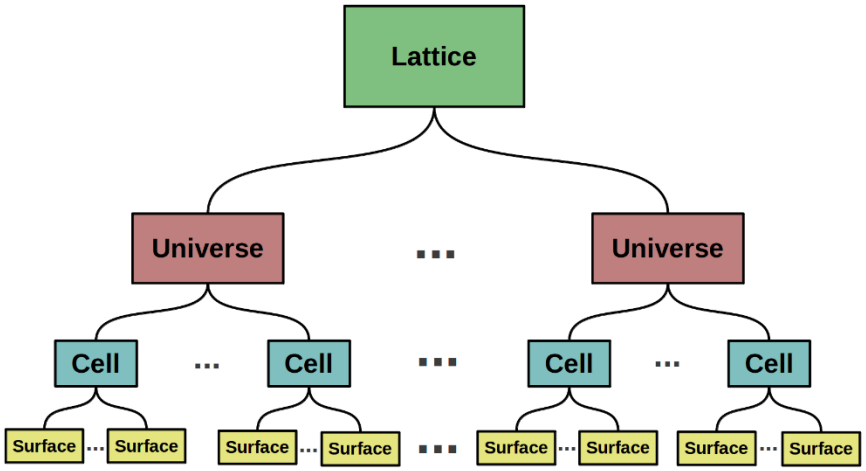

Figure 1: OpenMOC geometry hierarchy (https://mit-crpg.github.io/OpenMOC/_images/csg-primitives-hierarchy.png).

A cell is defined to be the region bounded by a boolean combination of surface halfspaces. Presently, OpenMOC only permits the use of halfspace intersections and does not support other boolean operations such as unions and differences. The region deined by the cell is subsequently filled by either a material or a universe.

A universe is a collection of one or more cells that fill the entirety of the xy-plane. Each cell may be filled with a material or a separate universe. Universes allow unique structures to be created from cells, and for simple replication of that structure throughout a model by placing it in various locations throughout the geometry.

Lattices are useful in the case for reactor cores which typically contain rectangular or hexagonal arrays of fuel pins. OpenMOC currently only contains a single lattice implementation for 2D Cartesian arrays. Each lattice is uniquely specified by the number of array elements along the x and y axes, the width and height of each lattice cell, and the universe to fill each lattice cell. The lattice specification represents a coordinate transformation such that the center of each lattice cell maps to the origin of the universe within it.

OpenMOC uses multi-group macroscopic nuclear cross sections, provided by the user. It does not perform self-shielding or depletion calculations which must be provided by preprocessing microscopic data with other codes like NJOY.

OpenMOC includes a module for importing nuclear data cross sections from an HDF5 binary file (http://www.hdfgroup.org/HDF5/). The multi-group cross sections to assign by material or cell must be defined as an HDF5 group with a string name or integer ID to identify the material or cell. The material group must contain the following floating point HDF5 datasets of multi-group cross section data:

- 'total' or 'transport' or 'nu-transport'
- 'nu-scatter matrix' or 'scatter matrix'
- 'chi'
- 'nu-fission'
- 'fission'

Each dataset should be a 1D array of floating-point values ordered by increasing energy group (*i.e.*, from highest to lowest energies). This includes the scattering matrix which should be inner strided by outgoing energy group and outer strided by incoming energy group.

1.7. The NJOY preprocessing code

The NJOY Nuclear Data Processing System is a modular computer code used for converting evaluated nuclear data in the ENDF format into libraries useful for applications calculations. NJOY handles a wide variety of nuclear effects, including resonances, Doppler broadening, heating (KERMA), radiation damage, thermal scattering, gas production, neutrons and charged particles, photo-atomic interactions, photonuclear reactions, self shielding, probability tables, photon production, and high-energy interactions (to 150 MeV). Output can include printed listings, special library files for applications, and Postscript graphics (https://t2.lanl.gov/nis/njoy/index.html).

The original NJOY code was written in FORTRAN77 and is not open source. It was later converted into FORTRAN-90 and is now available as NJOY21 as open source and written in C++ (http://www.njoy21.io/).

The most important modules of NJOY are [27]:

- **NJOY**: directs the flow of data through the other modules. Subsidiary modules for locale, ENDF formats, physics constants, utility routines, and math routines are grouped with the NJOY module for descriptive purposes

- **RECONR**: reconstructs pointwise (energy-dependent) cross sections from ENDF resonance parameters and interpolation schemes

- **BROADR**: Doppler-broadens and thins pointwise cross sections

- **UNRESR**: computes effective self-shielded pointwise cross sections in the unresolved energy range

- **THERMR**: produces cross sections and energy-to-energy matrices for free or bound scatterers in the thermal energy range

- **GROUPR**: generates self-shielded multigroup cross sections, group-to-group scattering matrices, photon production matrices, and charged-particle multigroup cross sections from pointwise input

- **ERRORR**: computes multigroup covariance matrices from pointwise covariance data.

Each module reads some data from one or more input files, transforms it, and writes the results on one or more output files. The order that the modules are used and the particular input and output files used are specified as in the following example:

```
[copy an ENDF file onto "tape20"]

  reconr
  20 21
    ...specific input for RECONR...
  broadr
  20 21 22
    ...specific input for BROADR...
  groupr
  20 22 0 23
    ...specific input for GROUPR...
  stop

  [copy tape24 to the DTF library]
```

In this case, RECONR reads the original ENDF tape and reconstructs the resonances and nonlinear interpolation schemes to prepare a PENDF (pointwise ENDF) tape on unit 21. BROADR reads the PENDF tape, Doppler broadens the pointwise cross sections, and writes a new temperature dependent PENDF tape on unit 22. Sometimes it needs some data from the ENDF tape, and this is why tape20 is also provided as input. Next, the information from the ENDF and PENDF tapes is run through the multigroup averaging process and written in GENDF format (a special groupwise variation of the ENDF format) on tape23.

2. Cross section homogenization with OpenMOC

In this project a fuel assembly geometry for a typical pressurized water reactor (PWR) is considered. A horizontal plot of such an assembly is shown in figure 2. It has 17x17-24 fuel rods and 20 guide tubes for control rod insertion. The fuel rod outer and inner cladding radius is 0.47 and 0.54 cm. The cladding material is considered standard Zircaloy-4. The fuel pellet outer radius is 0.47 cm. The pitch between fuel rods is x cm. The radius of the guide tubes is also 0.54 cm and their material is also Zircaloy.

The pellets are assumed to be of type UO2 with an enrichment of 4 w/o U235 and a density of 10.15 g/cm3. The water and Zircaloy density are 0.663 and 6.5 g/cm3. An average temperature of the fuel, the H2O moderator and the Zircaloy cladding of 1100 K and 600 K is assumed.

The pitch between two neighboring fuel assemblies is 21.42 cm. For the purpose of spectral calculations and hence cross section homogenization the fuel assemblies are assumed to be of infinite length in the z direction.

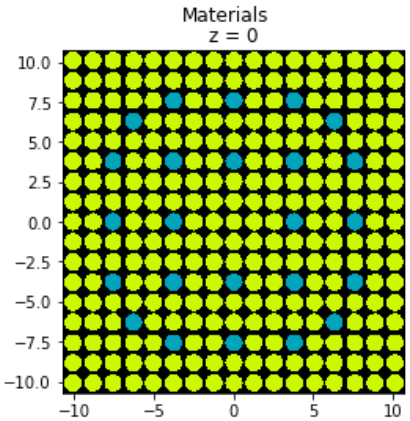

Figure 2: OpenMOC geometry plot of 17x17-24 fuel assembly

2.1. Input generation

The above described fuel assembly geometry can be created with OpenMOC with the following Python code:

```
# Create ZCylinder for the fuel and guide tubes
fuel_radius = openmoc.ZCylinder(x=0.0, y=0.0, radius=0.54)
guidetube_radius = openmoc.ZCylinder(x=0.0, y=0.0, radius=0.54)
```

```
# Create planes to bound the entire geometry
boundary = openmoc.RectangularPrism(21.42, 21.42)
boundary.setBoundaryType(openmoc.REFLECTIVE)
```

```
4.0%  UOX pin cell
uox40_cell = openmoc.Cell()
uox40_cell.setFill(materials['UO2'])
uox40_cell.setNumRings(3)
uox40_cell.setNumSectors(8)
uox40_cell.addSurface(-1, fuel_radius)

uox40 = openmoc.Universe(name='UO2')
uox40.addCell(uox40_cell)
```

```
# Guide tube pin cell
guide_tube_cell = openmoc.Cell()
guide_tube_cell.setFill(materials['Guide Tube'])
guide_tube_cell.setNumRings(3)
guide_tube_cell.setNumSectors(8)
guide_tube_cell.addSurface(-1, guide_tube_radius)

guide_tube = openmoc.Universe(name='Guide Tube')
guide_tube.addCell(guide_tube_cell)
```

```
# Moderator rings
moderator = openmoc.Cell()
moderator.setFill(materials['Water'])
moderator.addSurface(+1, fuel_radius)
moderator.setNumRings(3)
```

```
moderator.setNumSectors(8)
```

```
# Add moderator rings to each pin cell
pins = [uox40, guide_tube]
for pin in pins:
    pin.addCell(moderator)
```

```
# CellFills for the assembly
assembly1_cell = openmoc.Cell(name='Fuel Assembly')

assembly1 = openmoc.Universe(name='Fuel Assembly')
assembly1.addCell(assembly1_cell)
```

The 17x17 layout of the fuel and guide tube structure is done with the following code:

```
# A UO2 fuel assembly
assembly = openmoc.Lattice(name='UOX Assembly')
assembly.setWidth(width_x=1.26, width_y=1.26)

# Create a template to map to pin cell types
template = [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
            [1, 1, 1, 1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1, 1, 1, 1],
            [1, 1, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 1],
            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
            [1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1],
            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
            [1, 1, 4, 1, 1, 4, 1, 1, 1, 1, 1, 4, 1, 1, 4, 1, 1],
            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
            [1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1],
            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
```

```
            [1, 1, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 1],
            [1, 1, 1, 1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1, 1, 1, 1],
            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

universes = {1 : uox40,
             4 : guide_tube}

for i in range(17):
    for j in range(17):
        template[i][j] = universes[template[i][j]]

assembly.setUniverses([template])
```

```
# Root Cell/Universe
root_cell = openmoc.Cell(name='Full Geometry')
root_cell.setFill(assembly)
root_cell.setRegion(boundary)

root_universe = openmoc.Universe(name='Root Universe')
root_universe.addCell(root_cell)
```

The OpenMOC solver is finally initialized with the following commands:

```
cmfd = openmoc.Cmfd()
cmfd.setSORRelaxationFactor(1.5)
cmfd.setLatticeStructure(17,17)
cmfd.setGroupStructure([[1,2,3], [4,5,6,7]])
cmfd.setKNearest(3)
```

```
geometry = openmoc.Geometry()
geometry.setRootUniverse(root_universe)
```

```
geometry.setCmfd(cmfd)
```

```
track_generator = openmoc.TrackGenerator(geometry, num_azim, azim_spacing)
track_generator.setNumThreads(num_threads)
track_generator.generateTracks()
```

```
solver = openmoc.CPUSolver(track_generator)
solver.setConvergenceThreshold(tolerance)
solver.setNumThreads(num_threads)
solver.computeEigenvalue(max_iters)
```

The command "`solver.computeEigenvalue`" starts the eigenvalue determination with the MOC method.

2.2. Ensemble calculation loop

For the execution of the Total Monte Carlo method an HDF5 [28] library has been prepared which contains all the revenant cross sections. In this work a set of 100 variations of the best-estimate data has been prepared with the help of covariance data as explained in section 4. The HDF5 library contains material sections of the kind "UO2-xx" and "Moderator-xx" where "xx" runs from 1 to 100.

For the purpose of running the global loop of 100 spectral calculations a fuel assembly class has been created which is instantiated individually for each available cross section data set.

The HDF5 file is opened and the materials are defined with the following python code:

```
materials = load_from_hdf5(filename='data-ensemble-uo2.h5', directory='..')
```

```
Nmat = 100 # number of UO2 pin materials in the ensemble
uox40_array = []
```

```
gd_array = []

for n in range(Nmat):
    uo2_name = 'UO2-' + str(n)
    uox40_cell = openmoc.Cell()
    uox40_cell.setFill(materials[uo2_name])
    uox40_cell.setNumRings(3)
    uox40_cell.setNumSectors(8)
    uox40_cell.addSurface(-1, fuel_radius)

    uox40 = openmoc.Universe(name=uo2_name)
    uox40.addCell(uox40_cell)
    uox40_array.append(uox40)
```

The fuel assembly class is coded as follows:

```
class FuelAssembly:
    def __init__(self):
        # define fuel assembly basic parameters
        # CellFills for the assembly
        self.assembly1_cell = openmoc.Cell(name='Fuel Assembly')
        self.assembly1 = openmoc.Universe(name='Fuel Assembly')
        self.assembly1.addCell(self.assembly1_cell)

        # A mixed enrichment PWR MOX fuel assembly
        self.assembly = openmoc.Lattice(name='UOX Assembly')
        self.assembly.setWidth(width_x=1.26, width_y=1.26)

        # Create a template to map to pin cell types
        self.template = [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                         [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                         [1, 1, 1, 1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1, 1, 1, 1],
                         [1, 1, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 1],
                         [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                         [1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1],
                         [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
```

```
                        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                        [1, 1, 4, 1, 1, 4, 1, 1, 1, 1, 1, 4, 1, 1, 4, 1, 1],
                        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                        [1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1],
                        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                        [1, 1, 1, 4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 1],
                        [1, 1, 1, 1, 1, 4, 1, 1, 4, 1, 1, 4, 1, 1, 1, 1, 1],
                        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
        self.root_cell=[]
        self.root_universe=[]

    def setUniverse(self, uox40, guide_tube):
        universes = {1 : uox40, 4 : guide_tube}
        # loop over all pins
        for i in range(17):
            for j in range(17):
                self.template[i][j] = universes[self.template[i][j]]
        # write pin layout into assembly structure
        self.assembly.setUniverses([self.template])

    def setRootUniverse(self, boundary):
        # Root Cell
        self.root_cell = openmoc.Cell(name='Full Geometry')
        self.root_cell.setFill(self.assembly)
        self.root_cell.setRegion(boundary)
        # Root Universe
        self.root_universe = openmoc.Universe(name='Root Universe')
        self.root_universe.addCell(self.root_cell)

    def getRootUniverse(self):
        return self.root_universe
```

After each spectral calculation a mesh is used to extract the homogenized cross sections:

```
# Aggregate the total cross sections for each Material
# into a dictionary to pass to the mesh tally
domains_to_coeffsF = {}
domains_to_coeffsT = {}
domains_to_coeffsSx7 = {}
domains_to_coeffsPhi = {}

# Create OpenMOC Mesh on which to tally reaction rates
mesh = openmoc.process.Mesh()
mesh.dimension = [17*2, 17*2]
mesh.lower_left = [-21.42/2, -21.42/2]
mesh.upper_right= [ 21.42/2,  21.42/2]
mesh.width = [21.42/(17*2), 21.42/(17*2)]

# Array to store macroscopic XS results
Fission_results=[]
TotalXS_results=[]
Phi_results=[]

for n in range(Nmat):
    ##
    # . . . do the spectral calculation
    ##
    # Retrieve the Materials and number of groups from the geometry
    #
    materials = geometry.getAllMaterials()
    num_groups = geometry.getNumEnergyGroups()

    # Get microscopic cross sections
    for material_id in materials:
        domains_to_coeffsT[material_id] = np.zeros(num_groups)
        domains_to_coeffsF[material_id] = np.zeros(num_groups)
        domains_to_coeffsPhi[material_id] = np.zeros(num_groups)
        for group in range(num_groups):
            domains_to_coeffsT[material_id][group] = \
                                materials[material_id].getSigmaTByGroup(group+1)
            domains_to_coeffsF[material_id][group] = \
                                materials[material_id].getSigmaFByGroup(group+1)
            domains_to_coeffsPhi[material_id][group] = 1.0 # gives the flux
```

```
# Retrieve macroscopic cross sections
T_rates = mesh.tally_on_mesh(solver, domains_to_coeffsT, \
                                domain_type='material', volume='integrated')
F_rates = mesh.tally_on_mesh(solver, domains_to_coeffsF, \
                                domain_type='material', volume='integrated')
Phi     = mesh.tally_on_mesh(solver, domains_to_coeffsPhi, \
            domain_type='material', energy='by_group', volume='integrated')

Fission_results.append(F_rates)
TotalXS_results.append(T_rates)
Phi_results.append(Phi)
print('Case :', n, 'finished. ----------------------------------------------')
```

In the above code a rectangular mesh is defined. The '`solver`' has all results stored per FSR. The function call '`mesh.tally_on_mesh`' iterates over all FSRs and assigns results to the mesh cells in which they lie. The arrays '`T_rates`' then contain the reaction rates per mesh cell, the array '`Phi`' contains the corresponding neutron flux. The homogenized cross section per fuel assembly can be derived by either using a single mesh cell per fuel assembly or by calculating the weighted sum of reaction rates over many mesh cells.

3. Cross section preparation with NJOY

3.1. RECONR, BROADR, PURR, THERMR reconstruction

The ENDF/B-VII.1 datafile has been used for this project's analysis. In order to reconstruct the resonance part of the cross sections the RECONR module is applied first. Then the BROADR module is used to provide the Doppler broadening. In principle NJOY can calculate cross sections for different temperatures and many other boundary conditions in a single run. However, in this work an independent NJOY calculation has been used for every single set of boundary conditions in order to make the reading of the output file with an auxiliary Python script as simple as possible. The following sample NJOY input shows the reconstruction of the U235 cross sections:

```
echo 'NJOY U235 processing'
cp ./n-092_U_235.endf tape20
cat>input <<EOF
 -- U_235 Process U-235 (run WIMSR with FP yields)
 moder  /  Convert data to binary on Unit-21
 1 -21
 'ENDF/B-VII.1  U-235' /
 20 9228
 0 /
 reconr / Reconstruct x-sect from resonance parameters on Unit-22
 -21 -22
 'PENDF TAPE FOR U-235 FROM ENDF/B-VII.1   '/
 9228 2 /
 0.001  0.  0.005/ Reconstruction 0.1% (0.5% max)
 '92-U-235 FROM ENDF/B-VII.1  '/
 ' PROCESSED BY NJOY-2016 '/
 0 /
 broadr / Doppler broaden to Unit-23
 -21 -22  -23
 9228 1  0  0  0 /
 0.001 /
 1100.
 0 /
 purr / Doppler broaden /self-shield URP data to Unit-24
 -21 -23 -24
 9228 1 10 20 100 0 0 /
 1100.
 1.E10 3.E4 8000. 4500. 2800. 1800. 1200. 800. 500. 200. /
 0 /
 thermr / Add thermal scattering data to Unit-26
 0 -24 -26
 0 9228 12 1 1 0 0 1 221 1
 1100.
 0.001  4.0
stop
EOF

./njoy<input
echo 'saving output and pendf files'
```

The above code is a shell script which pipes the NJOY input into a file called '`input`' and then executes '`njoy`'.

3.2. GROUPR and ERRORR calculation

For codes using a finite number of energy groups the GROUPR module of NJOY can be used to generate group cross sections. GROUPR has a number of built in flux weighting functions for the cross section averaging. In this project the '69 epri-cpm' energy group structure has been used together with the 'epri-cell lwr' a priori weighting flux.

In many cases of practical interest, the flux will have dips corresponding to the resonances of the nuclides in the reactor materials. The reaction rates will also be reduced (self-shielded) because of these resonance dips. GROUPR provides two methods to estimate the effect of this self-shielding: the Bondarenko method [29] and the flux calculator. The latter was used in this work for small energies. For systems which are sensitive to energies below 500 eV there are many broad resonances which cannot accurately been treated with the Bondarenko approach. Above this threshold NJOY per default uses the Bondarenko method.

In the Bondarenko method the group weighting flux for each isotope n is approximated as follows:

(42) $\phi^n(E) = \frac{C(E)}{(\sigma_0^n + \sigma_t^n(E))}$

The function C is a smooth function of E – and often 1/E plus fission spectrum is used. The background cross section σ_o^n is the sum of the total of all other total cross sections except of isotope n and is usually a function of energy group g: $\sigma_{0,g}^n$. This approximation is suitable if the resonances of n are small – which is the case in the higher energy ranges.

The group average cross section then is calculated as a function of temperature T and background σ_o :

(43) $\sigma_{x,g}^n(\sigma_o, T) = \int_g dE\, \sigma_x^n(E,T)\phi^n(E) / \int_g dE\, \phi^n(E)$

Finally, a self-consistency condition is placed on all averaged cross sections if there is a mixture of resonance materials (if the number density is ρ_n for isotope n):

(44) $\sigma_{0,g}^n = \frac{1}{\rho_n} \sum_{m \neq n} \rho_m\, \sigma_{x,g}^m(\sigma_{0,g}^m, T)$

Eq. (44) must be fulfilled for each isotope n and is determined iteratively together with (43).

If GROUPR is requested to perform a flux solution in the broad resonance range the user can set an upper energy limit below which the self-shielded flux will be determined with higher accuracy according to the following equations.

The explicit flux solution of the GROUPR module assumes an infinite, homogenous medium which consists of two materials: one with almost constant “background” cross section σ_0 (representing all the nuclide except the resonance scatterer) and another one with resonance cross sections. Also, isotropic scattering in the center of mass system is assumed. The Boltzmann equations then has the following form:

(45) $\left(\sigma_0 + \sigma_t(E)\right)\phi(E) = C(E)\sigma_0 + \int_E^{E/\alpha} dE'\phi(E')\,\sigma_s(E') / (1-\alpha)E'$

The maximum energy loss per elastic scattering event at the resonance nuclide is E/α. The function C(E) is an unaffected (by the resonance scatterer), typical neutron flux of the reactor type considered (i.e. 1/E plus fission spectrum). It is a good approximation if the "back-ground" consists of a light scatterer which allows large energy losses per collision compared to the width of the resonances.

Eq. (45) is a good approximation for a homogenous material – but in reactor physics the het-erogeneity of the lattice requires some more modifications. A heterogeneity parameter β is defined:

(46) $\beta = \frac{V_f \sigma_e}{V_m \sigma_m}$

The volume ration of fuel and moderator zone is V_f/V_m.The escape cross section σ_e is ap-proximated as the ratio of σ_0 of the fuel divided by the number density of the fuel. Eq. (45) can then be updated for the flux ϕ_f in the fuel:

(47) $\left(\sigma_0 + \sigma_t\ (E)\right)\phi_f(E) = (1-\beta)C(E)\sigma_0 + \int_E^{\frac{E}{\alpha_f}} dE'\,\phi_f(E')\,\sigma_{s,f}(E')/\left(1-\alpha_f\right)E' + \int_E^{\frac{E}{\alpha_m}} dE'\,\phi_f(E')\,\beta\sigma_0\ /(1-\alpha_m)E'$

In a typical UO2 lattice the potential scattering cross section is about 10.6 barn, alpha for the oxygen in the fuel is 0.77, alpha for hydrogen in the moderator is set to 10^{-7}, the background cross section for oxygen in fuel is set to 7.5 barn, beta is typically 0.4 and the fraction of the admixed moderator cross section (O, 3.75 barn) in the external moderator cross section (2H, 2×20barn) is approximately 0.086.

These considerations lead to the following input block for GROUPR for U235, for example:

```
groupr / Generate group averaged data on Unit-25
-21  -26  0  25
9228 9 0 -5  1  1  1  1
'92-U-235 FROM ENDF/B-VII.1  '/
1100.
1000.
400 10.6 5000 0 0 0.7768 7.5 .40 1.7E-7 0.086  / weight function options
3 /          Temperature 1100.K
3 221 / thermal xsec
3 452 / total fission yield
3 455 / delayed nubar
5 18  / prompt spectrum
5 455 / delayed spectra
6 /
6 221 / thermal scattering matrix
0 /
0 /
```

The ERRORR module uses the information in files 33 to 40 to generate covariance matrices between a cross section and itself and between different cross sections and between cross sections between different nuclides. For example, in the case of the 69 energy group structure used in this work, a 69x69 covariance matrix for the total, scattering and fission cross section will be determined. Also, the covariance matrix between the total and scattering matrix is available. In the scope of this work only the covariance matrix between total, scattering and fission between themselves have been used to generate the ensemble of cross sections for the Total Monte Carlo method.

The ERROR module is started for U235 with the following input sequence.

```
errorr / Process data with errorr module
-21 -26 25 33 /
9228 9 5 1 0 /
1 1100. /
0 33 /
```

The following figures show the mentioned cross sections together with their covariance matrices:

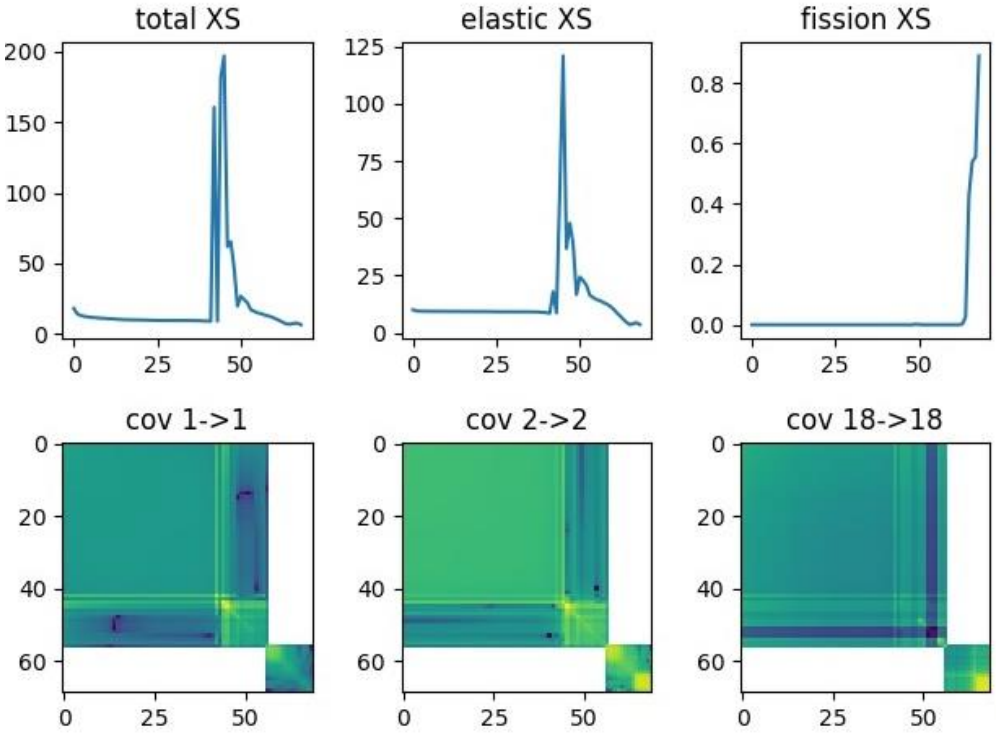

Figure 3: Total, elastic and fission cross section in 69 group structure from the GROUPR calculation for U238; covariance matrix between energy groups for total (1→1), elastic (2→2) and fission cross section (18→18).

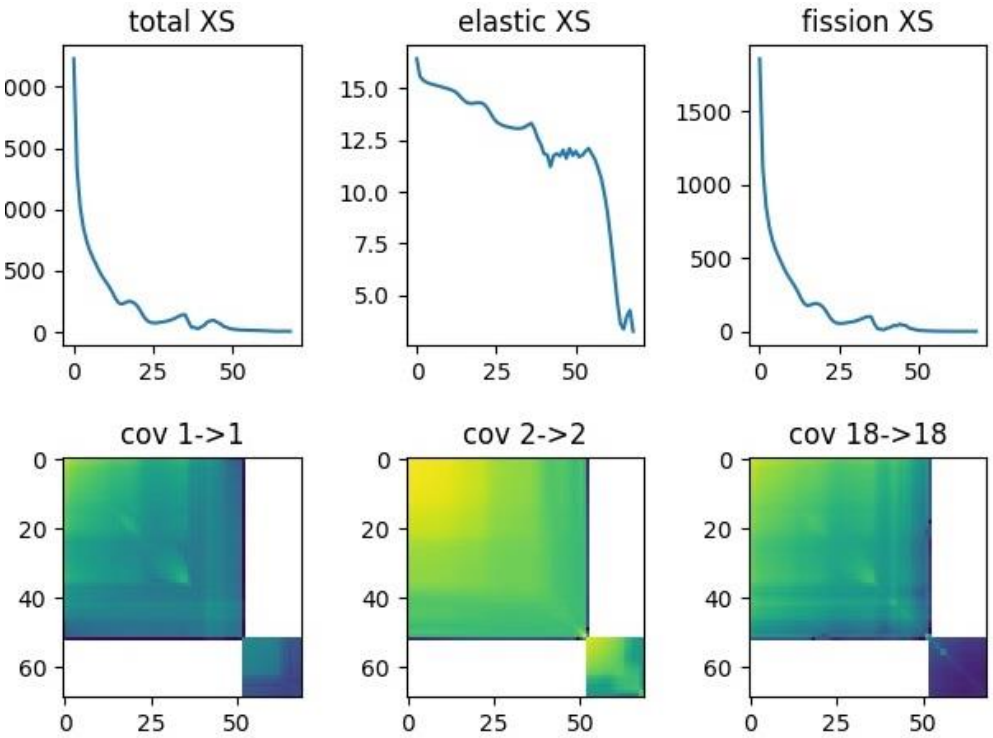

Figure 4: Total, elastic and fission cross section in 69 group structure from the GROUPR calculation for U235; covariance matrix between energy groups for total (1→1), elastic (2→2) and fission cross section (18→18).

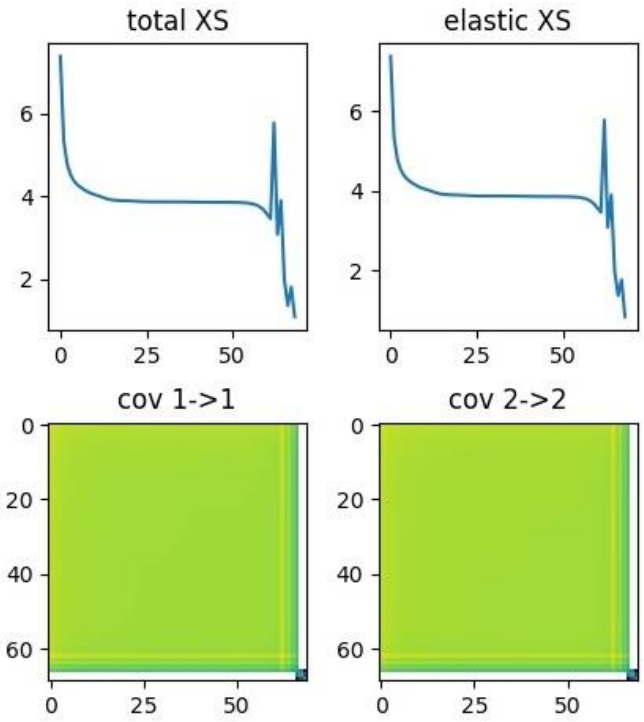

Figure 3: Total, elastic and fission cross section in 69 group structure from the GROUPR calculation for O16; covariance matrix between energy groups for total (1→1) and elastic (2→2) cross section.

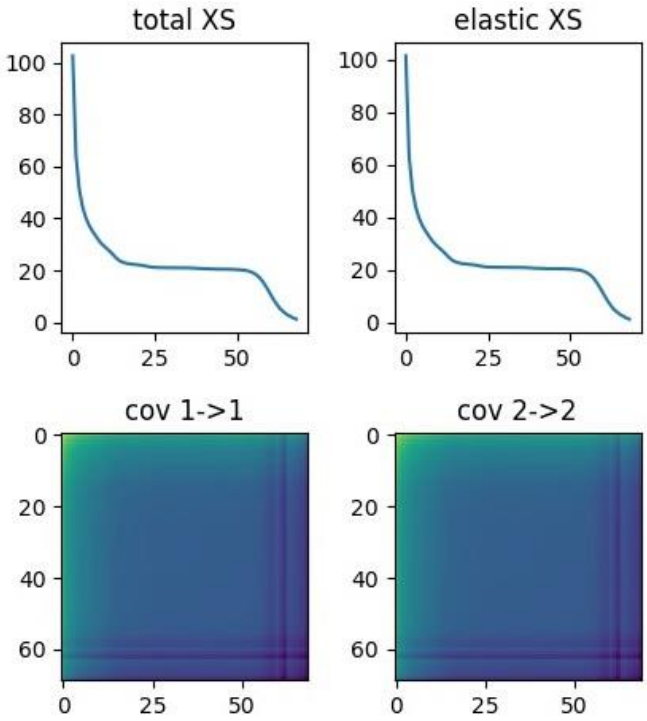

Figure 4: Total, elastic and fission cross section in 69 group structure from the GROUPR calculation for H01; covariance matrix between energy groups for total (1→1) and elastic (2→2) cross section.

3.3. OpenMOC cross section preparation

A Python script (readGROUPR.py, see appendix) was written to read the output of the GROUPR module and another script (readERORR.py, see appendix) for the extraction of the ERRORR output. These scripts store the cross section information in numpy (https://www.numpy.org/) arrays and safe them on disk.

A third Python sequence was written to combine this cross section information into the required material definitions for OpenMOC and save them in HDF5 format on disk for later use.

Initialization of HDF5 file:

```
Create the file to store C5G7 multi-groups cross-sections
f = h5py.File('XS-ensemble-uo2.h5')
f.attrs["# groups"] = 69

# Create a group to specify that MGXS are split by material (vs. cell)
material_group = f.create_group('material')
```

Reading saved output from GROUPR and ERRORR:

```
###############################################################################
# create UO2 data which mimicks convariance data from an ENDF file
# Later we use real data from ENDF, here we make just random variations
# to test the prcodure of random sampling
N = 100  # number of samples; MUST be equal with size of daten_h01_600 et al.
###############################################################################
# load microscopic cross sections generated from NJOY and GROUPR and ERRORR
#
daten_h01_600  =numpy.load('h01_600_numpy_ensemble.npz')   # total, elastic XS ense
mble yr1_1, yr2_2
daten_o16_600  =numpy.load('o16_600_numpy_ensemble.npz')   # total, elastic XS ense
mble yr1_1, yr2_2
daten_o16_1100 =numpy.load('o16_1100_numpy_ensemble.npz')  # total, elastic XS ense
mble yr1_1, yr2_2
daten_u235_1100=numpy.load('u235_1100_numpy_ensemble.npz') # total, elastic and fis
```

```
sion XS ensemble yr1_1, yr2_2, yr18_18

daten_u238_1100=numpy.load('u238_1100_numpy_ensemble.npz') # total, elastic and fis
sion XS ensemble yr1_1, yr2_2, yr18_18


daten_u235_1100_err=numpy.load('u235_1100_numpy_tables.npz') # energyGroups, totalX
S, elasticXS, fissionXS, elasticXSM, nubarXS, chiXS

daten_u238_1100_err=numpy.load('u238_1100_numpy_tables.npz') # energyGroups, totalX
S, elasticXS, fissionXS, elasticXSM, nubarXS, chiXS

daten_o16_1100_err =numpy.load('o16_1100_numpy_tables.npz')  # energyGroups, totalX
S, elasticXS, elasticXSM

daten_o16_600_err  =numpy.load('o16_600_numpy_tables.npz')   # energyGroups, totalX
S, elasticXS, elasticXSM

daten_h01_600_err  =numpy.load('h01_600_numpy_tables.npz')   # energyGroups, totalX
S, elasticXS, elasticXSM



h01_600_t_xs = daten_h01_600['arr_0']     # total cross section

h01_600_e_xs = daten_h01_600['arr_1']     # elastic cross section

h01_600_e_xsm= daten_h01_600_err['arr_3'] # elastic matrix


o16_600_t_xs = daten_o16_600['arr_0']     # total cross section

o16_600_e_xs = daten_o16_600['arr_1']     # elastic cross section

o16_600_e_xsm= daten_o16_600_err['arr_3'] # elastic matrix


o16_1100_t_xs = daten_o16_1100['arr_0']     # total cross section

o16_1100_e_xs = daten_o16_1100['arr_1']     # elastic cross section

o16_1100_e_xsm= daten_o16_1100_err['arr_3'] # elastic matrix


u235_1100_t_xs = daten_u235_1100['arr_0']  # total cross section

u235_1100_e_xs = daten_u235_1100['arr_1']  # elastic cross section

u235_1100_f_xs = daten_u235_1100['arr_2']  # fission cross section

u235_1100_e_xsm   = daten_u235_1100_err['arr_4']  # elastic matrix

u235_1100_f_nubar = daten_u235_1100_err['arr_5']  # nubar

u235_1100_f_chi   = daten_u235_1100_err['arr_6']  # chi


u238_1100_t_xs = daten_u238_1100['arr_0']  # total cross section

u238_1100_e_xs = daten_u238_1100['arr_1']  # elastic cross section

u238_1100_f_xs = daten_u238_1100['arr_2']  # fission cross section

u238_1100_e_xsm   = daten_u238_1100_err['arr_4']  # elastic matrix

u238_1100_f_nubar = daten_u238_1100_err['arr_5']  # nubar

u238_1100_f_chi   = daten_u238_1100_err['arr_6']  # chi
```

Definition of material constants:

```
density_h20 = 0.662 # g/cm3
density_uo2 = 10.15 # g/cm3
enr = 0.04 # 4% enrichment U235

uo2_m =  270.072  # g/mol molecular weight
h20_m =  18.015   # g/mol molecular weight
mol   = 6.022E+23  # 1 mol of particles
barn  = 1.0E-24    # cm2

nd_h20 = density_h20/h20_m  # mol/cm3
nd_uo2 = density_uo2/uo2_m  # mol/cm3

n_u235_fuel = nd_uo2*mol*enr         # atoms/cm3
n_u238_fuel = nd_uo2*mol*(1.0-enr)   # atoms/cm3
n_o_fuel    = nd_uo2*mol*2.0         # atoms/cm3
n_o_mod     = nd_h20*mol             # atoms/cm3
n_h_mod     = nd_h20*mol*2.0         # atoms/cm3

# microscopic cross sections are in barn
# macroscopic cross sections are in 1/cm
```

Creation of fuel and moderator cross sections:

```
xs_t_moderator = (n_h_mod * h01_600_t_xs + n_o_mod * o16_600_t_xs)*barn
xs_e_moderator = (n_h_mod * h01_600_e_xs + n_o_mod * o16_600_e_xs)*barn

# bring energy groups into right order, i.e. from high to low energy
xs_t_moderator = numpy.flip(xs_t_moderator, axis=0)
xs_e_moderator = numpy.flip(xs_e_moderator, axis=0)

xs_t_fuel = (n_o_fuel * o16_1100_t_xs + n_u238_fuel * u238_1100_t_xs + n_u235_fuel
* u235_1100_t_xs)*barn
```

```
xs_e_fuel = (n_o_fuel * o16_1100_e_xs + n_u238_fuel * u238_1100_e_xs + n_u235_fuel
* u235_1100_e_xs)*barn

xs_f_fuel = (                           n_u238_fuel * u238_1100_f_xs + n_u235_fuel
* u235_1100_f_xs)*barn


xs_t_fuel = numpy.flip(xs_t_fuel, axis=0)

xs_e_fuel = numpy.flip(xs_e_fuel, axis=0)

xs_f_fuel = numpy.flip(xs_f_fuel, axis=0)


chi_fuel   = (n_u235_fuel * u235_1100_f_chi + n_u238_fuel * u238_1100_f_chi)/(n_u23
5_fuel + n_u238_fuel)

nubar_fuel = (n_u235_fuel * u235_1100_f_nubar + n_u238_fuel * u238_1100_f_nubar)/(n
_u235_fuel + n_u238_fuel)


chi_fuel   = numpy.flip(chi_fuel, axis=0)

nubar_fuel = numpy.flip(nubar_fuel, axis=0)

nubar_fuel = nubar_fuel[:,1]


chi_fuel   = chi_fuel.flatten()

nubar_fuel = nubar_fuel.flatten()


xsm_e_fuel = (n_o_fuel * o16_1100_e_xsm + n_u238_fuel * u238_1100_e_xsm + n_u235_fu
el * u235_1100_e_xsm)*barn

xsm_e_moderator = (n_h_mod * h01_600_e_xsm + n_o_mod * o16_600_e_xsm)*barn


xsm_e_fuel = numpy.flip(xsm_e_fuel,axis=0)

xsm_e_fuel = numpy.flip(xsm_e_fuel,axis=1)


xsm_e_moderator = numpy.flip(xsm_e_moderator,axis=0)

xsm_e_moderator = numpy.flip(xsm_e_moderator,axis=1)
```

Generation of an ensemble of cross sections (100 in this example) according covariance information:

```
for n in range(100):

    #

    ###############################################################################

    ##############################      UO2             ############################

    ###############################################################################

    # Create a subgroup for UO2 materials data
```

```
    uo2_name = 'UO2-'+str(n)
    uo2 = material_group.create_group(uo2_name)

    # Total cross section -----------------------------------------------------
--
    # -----------------------------------------------------------------------
--
    sigma_t = xs_t_fuel[:,n]

    # Scattering cross section ------------------------------------------------
---
    # -----------------------------------------------------------------------
--
    # https://stackoverflow.com/questions/8904694/how-to-normalize-a-2-dimensional-
numpy-array-in-python-less-verbose
    sigma_s = normalize(xsm_e_fuel, axis=1, norm='l1')
    sigma_s = sigma_s * xs_e_fuel[:,n]

    # Fission cross section ---------------------------------------------------
--
    # -----------------------------------------------------------------------
--
    sigma_f = xs_f_fuel[:,n]

    # Nu fission cross section ------------------------------------------------
--
    # -----------------------------------------------------------------------
--
    nu_sigma_f = sigma_f * nubar_fuel
    chi = chi_fuel

    # Create datasets for each cross-section type
    uo2.create_dataset('total', data=sigma_t)
    uo2.create_dataset('scatter matrix', data=sigma_s)
    uo2.create_dataset('fission', data=sigma_f)
    uo2.create_dataset('nu-fission', data=nu_sigma_f)
    uo2.create_dataset('chi', data=chi)

    ###########################################################################
    ##############################      Guide Tube      ###########################
    ###########################################################################

    # Create a subgroup for guide tube materials data
```

```
    gd_name = 'Guide_Tube-' + str(n)
    guide_tube = material_group.create_group(gd_name)

    sigma_t = xs_t_moderator[:,n]

    sigma_s = normalize(xsm_e_moderator, axis=1, norm='l1')
    sigma_s = sigma_s * xs_e_moderator[:,n]

    sigma_f = numpy.zeros(69)
    nu_sigma_f = numpy.zeros(69)
    chi = numpy.zeros(69)

    # Create datasets for each cross-section type
    guide_tube.create_dataset('total', data=sigma_t)
    guide_tube.create_dataset('scatter matrix', data=sigma_s)
    guide_tube.create_dataset('fission', data=sigma_f)
    guide_tube.create_dataset('nu-fission', data=nu_sigma_f)
    guide_tube.create_dataset('chi', data=chi)

    ##############################################################################
    ##################################      Water      ###########################
    ##############################################################################

    # Create a subgroup for water materials data
    h20_name = 'Water-' + str(n)
    water = material_group.create_group(h20_name)

    sigma_t = xs_t_moderator[:, n]

    sigma_s = normalize(xsm_e_moderator, axis=1, norm='l1')
    sigma_s = sigma_s * xs_e_moderator[:, n]

    sigma_f = numpy.zeros(69)
    nu_sigma_f = numpy.zeros(69)
    chi = numpy.zeros(69)

    # Create datasets for each cross-section type
    water.create_dataset('total', data=sigma_t)
    water.create_dataset('scatter matrix', data=sigma_s)
```

```
    water.create_dataset('fission', data=sigma_f)
    water.create_dataset('nu-fission', data=nu_sigma_f)
    water.create_dataset('chi', data=chi)

# Close the hdf5 data file
f.close()
```

4. Total Monte Carlo results

Weit hinten, hinter den Wortbergen, fern der Länder Vokalien und Konsonantien leben die Blindtexte. Abgeschieden wohnen Sie in Buchstabhausen an der Küste des Semantik, eines großen Sprachozeans. Ein kleines Bächlein namens Duden fließt durch ihren Ort und versorgt sie mit den nötigen Regelialien. Es ist ein paradiesmatisches Land, in dem einem gebratene Satzteile in den Mund fliegen. Nicht einmal von der allmächtigen Interpunktion werden die Blindtexte beherrscht – ein geradezu unorthographisches Leben.

5. Conclusions

One important task for future work is to extend the OpenMOC code with the Bateman equations to generate cross sections as a function of burnup. In commercial spectral code applications cross section tables with burnup steps of about 5 MWd/kgU are required. Also, for typical reactor core simulations different temperatures and control rod states must be taken into account. This leads to a tree of pre-calculated, homogenized cross sections as exemplified in figure x.

6. References

[1] W. Boyd et al., The OpenMOC method of characteristics neutral particle transport code, Annals of Nuclear Energy, Volume 68, June 2014, Pages 43-52

[2] X-5 Monte Carlo Team, “MCNP - A General N-Particle Transport Code, Version 5” Volume I: Overview and Theory, LA-UR-03-1987

[3] P.K. Romano et al., OpenMC: A state-of-the-art Monte Carlo code for research and development, Annals of Nuclear Energy, Volume 82, August 2015, Pages 90-97

[4] L. Qiao, Y. Zheng, C. Wan, Uncertainty quantification of sodium-cooled fast reactor based on the UAM-SFR benchmarks: From pin-cell to full core, Annals of Nuclear Energy, Volume 128, June 2019, Pages 433-442

[5] B. Batki, A. Kereszturi, I. Panka, Calculation of core safety parameters and uncertainty analyses during unprotected transients for the ALLEGRO and a sodium-cooled fast reactor, Annals of Nuclear Energy, Volume 118, August 2018, Pages 260-271

[6] E. Bauge, Full Model Nuclear Data and Covariance Evaluation Process Using TALYS, Total Monte Carlo and Backward-forward Monte Carlo, Nuclear Data Sheets, Volume 123, January 2015, Pages 201-206

[7] E. Alhassan et al., Combining Total Monte Carlo and Benchmarks for Nuclear Data Uncertainty Propagation on a Lead Fast Reactor's Safety Parameters, Nuclear Data Sheets, Volume 118, April 2014, Pages 542-544

[8] P. Sabouri, Propagation of Nuclear Data Uncertainties in Deterministic Calculations: Application of Generalized Perturbation Theory and the Total Monte Carlo Method to a PWR Burnup Pin-Cell, Nuclear Data Sheets, Volume 118, April 2014, Pages 523-526

[9] NVIDIA Tesla V100 GPU Architecture Whitepaper, WP-08608-001, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[10] R.E. MacFarlane, The NJOY Nuclear Data Processing System, LA-UR-12-27079

[11] R.E. MacFarlane, A.C. Kahler, Methods for Processing ENDF/B-VII with NJOY, Nuclear Data Sheets Volume 111, Issue 12, December 2010, Pages 2739-2890

[12] J. Duderstadt, J. Hamilton, Nuclear reactor analysis, Wiley, 1976

[13] P. Reuss, Neutron physics, EDP Sciences, 2008

[14] Y. Azmy (editor), Nuclear computational science: a century in review, Chapter 4, Springer 2010

[15] K. Smith, An analytic nodal method for solving the two-group, multidimensional, static and transient neutron diffusion equations, Thesis MIT, 1979

[16] H. Bateman, The solution of differential equations occurring in the theory of radioactive transformations, Proc. Cambirdge Phil. Soc. 15, 423 (1910)

[17] D.G. Cacuci (editor), Handbook of nuclear engineering, Vol. 1, Chapter 5, Springer 2010

[18] M.B, Chadwick et al., ENDF/B-VII.1 Nuclear Data for Science and Technology: Cross Sections, Covariances, Fission Product Yields and Decay Data, Nuclear Data Sheets Volume 112, Issue 12, December 2011, Pages 2887-2996

[19] https://www.oecd-nea.org/dbdata/jeff/jeff33/index.html

[20] ENDF-6 Formats manual, report BNL-90365-2009 (2009)

[21] https://www.njoy21.io/NJOY2016/

[22] K. Koebke, M.R. Wagner, The determination of the pin power distribution in a reactor core on the basis of nodal coarse mesh calculations, Atomkernenergie Bd. 30 (1977), p. 136-142

[23] H.D. Fischer, H. Finnemann, The nodal integration method – a diverse solver for neutron diffusion problems, Atomkernenergie Bd. 39 (1981), p. 229-234

[24] W. R. D. Boyd, Massively Parallel Algorithms for Method of Characteristics Neutral Particle Transport on Shared Memory Computer Architectures." M.S. Thesis, MIT (2014)

[25] A. Cholesky, Sur la résolution numérique des systèmes d'équations linéaires, https://journals.openedition.org/sabix/529

[26] T. Soyata, GPU Parallel Program Development Using CUDA, Chapman and Hall, 2018

[27] https://t2.lanl.gov/nis/njoy/njoy01.html

[28] https://portal.hdfgroup.org/display/HDF5/HDF5

[29] C.L. Dembia, G.D. Recktenwald, M.R. Deinert, Bondarenko method for obtaining group cross sections in a multi-region collision probability model, Progress in Nuclear Energy Volume 67, August 2013, Pages 124-131