YAKXS – The XML Multigroup Cross Section Library

Yaqi Wang, Javier Ortensi

March 28, 2016

1 Introduction

A multigroup cross section library in a designed format is required to manage all isotopes at all state points for the depletion calculation. A state point is described by several state variables such as the temperatures of fuel and coolant, soluble boron concentration, accumulated burnup, etc. Existing library formats like ANISN [1], ISOTXS [2], AMPX [3] created decades ago, are obscure to use or do not support all functionalities required by the transport calculations. There is a need to modernize these formats and extend their functionalities with the latest available computing tools. The successful use of XML (eXtendable Makeup Language) [4] format as the input for INSTANT [5] leads us to believe that XML should be used for our multigroup cross section library. This XML cross section library format is developed for the radiation transport library, YAK, within MOOSE [6]. It is therefore named as YAKXS. YAKXS is designed to provide a general toolkit for radiation transport calculation but not limited to be used only by YAK. It has several appealing features:

- It is in a general make-up language for which robust parsers exist;
- Data are managed hierarchically with the natural XML tree structure;
- The sequence of nodes in an XML file on the same tree level is irrelevant, which gives great flexibility to manage all state points and all isotopes;
- It is human readable although designed for computer driven data manipulation;
- It allows complete validity check;
- Comments can be inserted with the paired `'<!--'` and `'-->'`.

To save on storage, an XML library can be converted to a binary file with netCDF [7]. It needs to be noted beforehand that all symbols in a XML file are *case sensitive*. A manipulator of YAKXS in c++ with RapidXML [8] is created. A Fortran interface is going to be provided to facilitate coding efforts in Fortran. With this format, operations including interpolation and mixing can be done with a uniform interface. Users and developers are shielded from the complexity of these operations.

2 Terminology

For clarity, we define several terms which will be used frequently in this document.

1. *cross sections* are the generic name of all material properties required by neutronics calculation;
2. *reaction types* valid so far are include in Table 1. Following rule is applied to their MT number: if a reaction is in the ENDF [9] library, it has a valid positive MT number assigned, otherwise a negative number is used. More reaction types can be added when desired.
3. *cross section index* indicates the dependency of the cross section. They are collected in Table 2 where G is the number of energy groups, L is the order of angular anisotropy and I is the number of delayed neutron precursor groups;
4. *multigroup library* contains all sorts of cross sections of a collection of isotopes at all tabulated state points;
5. *master multigroup library* contains a number of multigroup libraries that provide the cross sections for a multi-region structure (i.e., assembly with multiple pins, pin with multiple regions, etc.) ;
6. *grid* is the grid formed by all state variables contained in the multigroup library. Dimension of the grid is up to the generator. For example, a grid may contains fuel temperature and burnup as its two dimensions;
7. *table* contains all sorts of cross sections of a collection of isotopes at a specific grid point or a state point;

Table 1 Valid reaction types in the multigroup library.

Identifier	Notation	MT number	Allowed class	Mixing type No.
Total	$\sigma_{t,g,l,m}$	1	ALL	1
Absorption	$\sigma_{a,g}$	27	ALL	1
Fission	$\sigma_{f,p}$	18	Undefined, Fissile, Fertile, OtherActinide	1
Removal	$\sigma_{r,g}$	-1	Undefined, Fissile, Fertile, OtherActinide	1
Transport	$\sigma_{tr,g,l,m}$	-2	ALL	1
Scattering	$\sigma_{s,g,p,l,m}$	-3	ALL	1
nuFission	$\nu\sigma_{f,p}$	-4	Undefined, Fissile, Fertile, OtherActinide	1
kappaFission	$\kappa\sigma_{f,g}$	-5	Undefined, Fissile, Fertile, OtherActinide	1
FissionSpectrum	$\chi_{g,p}$	-6	Undefined, Fissile, Fertile, OtherActinide	2
DNFraction	$\beta_{i,p}$	-7	Undefined, Fissile, Fertile, OtherActinide	2
DNSpectrum	$\chi_{d,i,g}$	-8	Undefined, Fissile, Fertile, OtherActinide	3
NeutronVelocity	$v_{g,l,m}$	-9	ALL	3
DNPlambda	λ_i	-10	Undefined, Fissile, Fertile, OtherActinide	3

Table 2 Valid reaction index.

Notation	Minimum	Maximum	Meaning
g	1	G	energy group index
p	1	G	secondary energy group index
l	0	L	polar spherical harmonics index
m	-l	l	azimuthal spherical harmonics index
i	1	I	delayed neutron precursor index

Table 3 Isotope classes.

Class name
Undefined
Fissile
Fertile
OtherActinide
FissionProduct
Structure
Coolant
Control

8. *isotope* contains all sorts of cross sections of a given isotope at a grid point. An isotope could be a isotope from the real word or a lumped pseudo-isotope. Recognized isotopes also have a default isotope class assigned;

9. *isotope class* is inherited from ISOTXS, the recognized isotope classes are included in Table 3.

   Not all reactions are valid of a specific isotope class. Only isotopes in class of Fissile, Fertile, OtherActinide and Undefined are allowed to have fission related cross sections as indicated in Table 1.

10. *library-wise cross sections* mean the cross sections which are fixed for all isotopes and all tables;

11. *table-wise cross sections* mean the cross sections which are fixed for all isotopes;

12. *mixing types* and their formulations are included in Table 4. All reactions have a mixing type which can be seen in Table 1.

Table 4 Mixing types.

No.	Mixing type	Formulation
1	Weighted_Summation	$\sum_{j=1}^{N_{iso}} N_j \sigma_{x,j}$
2	Weighted_Average	$\frac{\sum_{j,\sigma_{x,j}\neq 0} N_j \sigma_{x,j}}{\sum_{j,\sigma_{x,j}\neq 0} N_j}$
3	No_Weighting	σ_x

3 YAKXS - XML Format of the Master Multigroup Library

The XML format of the master multigroup library is defined in this section. The master multigroup library must contain at least one Multigroup_Cross_Section_Library. When the default value of an attribute or a node is not provided, it is required and must be specified in the library. Only a XML file following this format is considered a valid YAKXS master multigroup library.

The name (XML tag) of the root element is always Multigroup_Cross_Section_Libraries and its attributes are:

- *Name*

  Description: the name of the library

  Data type: string

  Default value: N/A

  Selection: any valid string

- *NGroup*

  Description: the number of energy groups G

  Data type: unsigned integer

  Default value: N/A

  Selection: must be greater than 0.

The name (XML tag) of the first element Multigroup_Cross_Section_Library defines a single multi-group library and its attributes are:

- *ID*

  Description: number of the material region to which the cross section are assigned

Data type: unsigned integer

Default value: N/A

Selection: must be greater than 0.

- *Description*

  Description: provides information

  Data type: string

  Default value: ""

  Selection: any valid string.

- *Ver*

  Description: the version of the multigroup library format

  Data type: string

  Default value: N/A

  Selection: 1.0 is the only valid value.

- *Generator*

  Description: the name of the generator of the library

  Data type: string

  Default value: 'INL'

  Selection: any valid string.

- *TimeCreated*

  Description: the time of the library creation

  Data type: string

  Default value: "

  Selection: any valid string.

- *InMeter*

  Description: to indicate if the cross section data is in unit of meter

  Data type: logical

  Default value: false

  Note: By default, the length unit of all cross sections is centimeter.

3.1 *Tabulation*

Description: names of all grid coordinates

Data type: string vector

Default value: N/A

Selection: valid strings

Note: Number of names must be greater or equal to 1. All names in Tabulation will be the requested node on the same level of Tabulation. They do not have default values; generator needs to provide a real vector for each of grid coordinate with non-zero length to create the grid. All the real vectors of all grid coordinates must be ordered ascendingly. Users can provide units for each grid coordinate by setting a string attribute *Unit* for the grid node. The default value of *Unit* will be an empty string.

3.2 *ReferenceGridIndex*

Description: reference index of all grid coordinates

Data type: integer vector

Default value: 1 for all grid coordinates

Note: The size of this integer vector must be equal to the number of grid coordinates in Tabulation. Each element in the vector is corresponding to a grid. The value of a grid index must be in between one and the number of the points of the corresponding grid coordinate.

3.3 *AllReactions*

Description: names of all reactions included in the library

Data type: string vector

Default value: N/A

Selection: valid reaction type identifiers

3.4 *LibrarywiseReactions*

Description: names of all reactions which do not have the state and isotope dependency

Data type: string vector

Default value: ″

Selection: valid reaction type identifiers

3.5 *TablewiseReactions*

Description: names of all reactions which do not have the isotope dependency

Data type: string vector

Default value: ″

Selection: valid reaction type identifiers

3.6 *Table*

A Table contains cross sections of all isotopes at one specific state point. There could be multiple Tables in the library. The maximum number of Tables in Multigroup_Cross_Section_Library is the product of the size of all grid coordinates. Tables differentiate with each other by their attribute gridIndex. Multigroup_Cross_Section_Library does not have to contain tables of all grid points.

- *gridIndex*
  Description: grid coordinate
  Data type: integer vector
  Default value: all ones
  Selection: each element of the coordinate vector is greater than or equal to 1 and less than or equal to the size of the corresponding grid coordinate.

3.6.1 *Isotope*

A Isotope contains cross sections of all reactions of one isotope at a specific state point. To have a clear explanation, all its possible reactions as child elements are included in a separate section Reaction. There could be multiple Isotopes in a Table. Isotopes differentiate with each other by their attribute Name. Pseudo isotopes are allowed, which are indicated by their names with a sub-string being *pseudo*. Two attributes MAT and Class are invalid for pseudo isotopes.

- *Name*

  Description: name of the isotope

  Data type: string

  Default value: N/A

  Note: Name of the isotope must be the element symbol (all capitalized) followed by the mass number, for example U235. Names containing *pseudo* are also valid, which is used to indicate that cross sections are only for a pseudo isotope.

- *MAT*

  Description: MAT number of the isotope

  Data type: integer

  Default value: the MAT number of the isotope

  Selection: Users can use this for checking the correctness of isotope Name. This attribute is not valid for pseudo isotopes.

- *L*

  Description: order of spherical harmonics expansion for scattering and possibly for total and transport cross sections L

  Data type: integer

  Default value: 0

  Selection: Any integer greater than 0. If L is equal to 0, the scattering of this isotope is isotropic.

- *Class*

  Description: class of the isotope

  Data type: string

  Default value: default class of the isotope

  Selection: Any valid isotope class. This attribute is not valid for pseudo isotopes.

- *I*

  Description: number of delayed neutron groups of the isotope

  Data type: integer

  Default value: 0

  Selection: Any integer greater than or equal to 0. if 0 is provided, no delayed neutron data will present for the isotope.

- *NS*

  Description: number of spatial expansions

  Data type: integer

  Default value: 1

  Selection: Any integer greater than 0. 1 means no spatial dependency.

3.6.2 *Tablewise*

A Tablewise element contains cross sections of all isotope at one specific state point. All possible reactions are included in a separate section Reaction. Tablewise cross sections are solely used to be filled in cross sections missing in the isotopes. Isotope cross sections take precedence over Tablewise cross sections of the same reactions when both exist.

- *L*

  Description: order of spherical harmonics expansion for scattering and possibly for total and transport cross sections L

  Data type: integer

  Default value: 0

  Selection: Any integer greater than 0. If L is equal to 0, the scattering of this isotope is isotropic.

- *I*

  Description: number of delayed neutron groups of the isotope

  Data type: integer

  Default value: 0

  Selection: Any integer greater than or equal to 0. if 0 is provided, no delayed neutron data will present for the isotope.

- *NS*

  Description: number of spatial expansions

  Data type: integer

  Default value: 1

  Selection: Any integer greater than 0. 1 means no spatial dependency.

3.7 *Librarywise*

Librarywise element contains all library-wise reaction cross sections which do not depend on the state and constant for all isotopes. All possible reactions are included in a separate section Reaction. Librarywise cross sections are solely used to be filled in cross sections missing in the isotopes. Isotope cross sections take precedence over Librarywise cross sections of the same reactions when both exist. Similarly, Tablewise cross sections take precedence over Librarywise cross sections of the same reactions when both exist.

- *L*

  Description: order of spherical harmonics expansion for scattering and possibly for total and transport cross sections L

  Data type: integer

  Default value: 0

  Selection: Any integer greater than 0. If L is equal to 0, the scattering of this isotope is isotropic.

- *I*

  Description: number of delayed neutron groups of the isotope

  Data type: integer

  Default value: 0

  Selection: Any integer greater than or equal to 0. if 0 is provided, no delayed neutron data will present for the isotope.

- *NS*

  Description: number of spatial expansions

  Data type: integer

  Default value: 1

  Selection: Any integer greater than 0. 1 means no spatial dependency.

3.8 Valid Reactions

This section describe the format of all valid reactions in Table 1. Allowed index of reactions are: g, p, i, m, l. Different combinations of index are used for different reactions. When a sample reaction $\sigma_{g,l}$ supports indices with 'gl', the data are provided in the order: $\sigma_{g,l}, g = 1, \cdots, G; l = 0, \cdots, L$. Similar rule applies for all kinds of combinations.

3.8.1 *Total*

Description: total cross sections

Data type: a vector of real numbers

Default value: N/A

Unit: cm^{-1}

Note: the size of the vector depends on the index. When index is 'g', it must be equal to the number of energy groups G. When index is 'gl', it must be equal to the number of energy groups G times the order of spherical harmonics expansion plus 1 $L+1$. When index is 'glm', it must be equal to $G \times (L+1) \times (L+1)$.

It has one attribute:

- *index*

  Description: subscript of the total cross section

  Data type: character(s)

  Default value: 'g'

  Selection: 'g'/'gl'/'gml' - 'g' means the total cross section only has the group dependency, i.e., all angular moments are the same. 'gl' means the total cross section also has the polar angle dependency. 'gml' means the total cross section has the full angle dependency.

3.8.2 *Fission*

Description: the fission cross sections

Data type: a vector of real numbers

Default value: N/A

Unit: cm^{-1}

Note: the size of the vector must be equal to the number of energy groups G. This cross section only valid for isotopes with classes of Fissile, Fertile, OtherActinides and Undefined in Table 3.

It has one attribute:

- *index*

  Description: subscript of the fission cross section

  Data type: character(s)

  Default value: ’g’

  Selection: ’g’ - ’g’ means the fission cross section only has the group dependency.

3.8.3 *Removal*

Description: the removal cross sections

Data type: a vector of real numbers

Default value: N/A

Unit: cm^{-1}

Note: the removal cross section is defined as $\sigma_{r,g} \equiv \sigma_{t,g} - \sigma_{s,0}^{g\to g}$. The size of the vector must be equal to the number of energy groups G.

It has one attribute:

- *index*

  Description: subscript of the removal cross section

  Data type: character(s)

  Default value: ’g’

  Selection: ’g’ - ’g’ means the removal cross section only has the group dependency.

3.8.4 *Transport*

Description: transport cross sections

Data type: a vector of real numbers

Default value: N/A

Unit: cm^{-1}

Note: transport cross section is defined as $\sigma_{tr,g} \equiv \sigma_{t,g,l,m} - \sum_{g'=1}^{G} \sigma_{s,l,m}^{g\to g'}$. The size of the vector depends on the index. When index is ’g’, it must be equal to the number of energy groups G. When index is ’gl’, it must be equal to the number of energy groups G times the order of spherical harmonics expansion plus 1 $L+1$. When index is ’glm’, it must be equal to $G \times (L+1) \times (L+1)$.

It has one attribute:

- *index*

  Description: subscript of the transport cross section

  Data type: character(s)

  Default value: ’g’

  Selection: ’g’/’gl’/’gml’ - ’g’ means the transport cross section only has the group dependency, i.e., all angular moments are the same. ’gl’ means the transport cross section also has the polar angle dependency. ’gml’ means the transport cross section has the full angle dependency.

3.8.5 *Scattering*

It has three attribute:

- *index*

  Description: subscript of the scattering cross section

  Data type: character(s)

  Default value: 'pgl'

  Selection: 'pgl'/'pgml' - 'pgl' means the scattering cross section only has the group dependency and rotational irrelevant. 'pgml' means the scattering cross section has the full angle dependency.

- *profile*

  Description: profiling index

  Data type: integer

  Default value: 0

  Selection: 0/1 - no profiling/has profiling.

  Note: *Profile* is designed to save the input effort and the size of the file containing the scattering cross sections. There are $G \times (L+1)$ pairs of integers. All these pairs are ordered by energy index g first then anisotropy group n, i.e. $((g = 1, G), n = 0, L)$. Each pair has the first departure scattering group and the last departure scattering group for the anisotropy n and the group g. *Profile* is used to reduce the amount of inputs for the scattering matrix. For example, the following 7-by-7 scattering matrix where non-zeros are marked with $\times$,

$$\begin{array}{c} \rightarrow g' \\ \downarrow g \begin{bmatrix} \times & & & & & & \\ \times & \times & & & & & \\ \times & \times & \times & & & & \\ & & \times & \times & \times & & \\ & & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times & \times \end{bmatrix}_{7\times 7} \end{array}$$

  has *Profile* $[1 \quad 1 \qquad 1 \quad 2 \qquad 1 \quad 3 \qquad 3 \quad 5 \qquad 3 \quad 7 \qquad 3 \quad 7 \qquad 3 \quad 7]$.

- *has2l* Description: if the cross section contains the $2l+1$ coefficients

  Data type: logical

  Default value: false

Depending on the value of its profile attribute, it has different contents.

When profile is equal to 0, it is a *leaf* element.

- Description: scattering cross sections

  Data type: a vector of real numbers

  Default value: N/A

  Unit: cm^{-1}

  Note: the size of the vector must be equal to the square the number of energy groups if its profile attribute is set to 0.

When profile is equal to 1, it contains two *subelements*.

- *Profile*

  Description: profiling vector

  Data type: a vector of integer numbers

  Default value: N/A

  Note: the size of the vector must be equal to two times the number of energy groups G times the order of scattering anisotropy plus 1 $L+1$ for index 'pgl' and $G \times (L+1) \times (L+1)$ for index 'pgml'. The meaning of the vector is explained in profile.

- *Value*

  Description: part of values of the scattering matrices

  Data type: a vector of real numbers

  Default value: N/A

  Unit: cm^{-1}

  Note: the size of the vector must be equal to the summation of all the differences of paired profile values.

3.8.6 *nuFission*

Description: ν times the fission cross sections

Data type: a vector of real numbers

Default value: N/A

Unit: cm^{-1}

Note: the size of the vector must be equal to the number of energy groups G. This cross section only valid for isotopes with classes of Fissile, Fertile, OtherActinides and Undefined in Table 3.

It has one attribute:

- *index*

  Description: subscript of the ν fission cross section

  Data type: character(s)

  Default value: 'p'

  Selection: 'p' - 'p' means the ν fission cross section only has the group dependency.

3.8.7 *kappaFission*

Description: κ times the fission cross sections

Data type: a vector of real numbers

Default value: N/A

Unit: J/cm

Note: the size of the vector must be equal to the number of energy groups G. This cross section only valid for isotopes with classes of Fissile, Fertile, OtherActinides and Undefined in Table 3.

It has one attribute:

- *index*

  Description: subscript of the κ fission cross section

  Data type: character(s)

  Default value: 'g'

  Selection: 'g' - 'g' means the κ fission cross section only has the group dependency.

3.8.8 *FissionSpectrum*

Description: fission spectrum

Data type: a vector of real numbers

Default value: N/A

Unit: dimensionless

Note: the size of the vector must be equal to the number of energy groups G. The summation of the fission neutron spectrum for all energy groups must be equal to one. When the number of delayed neutron groups I in Isotope or Tablewise or Librarywise is non-zero, the fission spectrum given should be the prompt fission spectrum, otherwise it should be the averaged neutron spectrum.

It has one attribute:

- *index*

  Description: subscript of the total cross section

  Data type: character(s)

  Default value: 'g'

  Selection: 'g'/'pg' - 'g' means the fission spectrum only has the group dependency. 'pg' means the fission spectrum depends on the energy of the neutron introducing the fission event.

3.8.9 *DNFraction*

Description: delayed neutron fraction

Data type: a vector of real numbers

Default value: N/A

Unit: dimensionless

Note: the size of the vector depends on the index. When it is equal to 'i', the size must be equal to the number of delayed neutron precursor groups I. When it is equal to 'pi', the size must be equal to the number of delayed neutron precursor groups I times the number of energy groups G. This cross section only valid for isotopes with classes of Fissile, Fertile, OtherActinides and Undefined in Table 3.

It has one attribute:

- *index*

  Description: subscript of the delayed neutron fraction

  Data type: character(s)

  Default value: 'i'

  Selection: 'i'/'pi' - 'i' means the delayed neutron fractions only have the delayed neutron group dependency and 'pi' means fractions also depends on the neutron energy.

3.8.10 *DNSpectrum*

Description: delayed neutron spectrum

Data type: a vector of real numbers

Default value: N/A

Unit: dimensionless

Note: the size must be equal to the number of delayed neutron precursor groups I times the number of energy groups G. This cross section only valid for isotopes with classes of Fissile, Fertile, OtherActinides and Undefined in Table 3.

It has one attribute:

- *index*

  Description: subscript of the delayed neutron spectrum

  Data type: character(s)

  Default value: 'gi'

  Selection: 'gi' means delayed neutron spectrum always depends on the delayed neutron group index and the emitted neutron energy.

3.8.11 *NeutronVelocity*

Description: averaged neutron velocities

Data type: a vector of real numbers

Default value: N/A

Unit: cm/s

Note: the size of the vector depends on the index. When index is 'g', it must be equal to the number of energy groups G. When index is 'gl', it must be equal to the number of energy groups G times the order of spherical harmonics expansion plus 1 $L+1$. When index is 'glm', it must be equal to $G \times (L+1) \times (L+1)$.

It has one attribute:

- *index*

  Description: subscript of the total cross section

  Data type: character(s)

  Default value: 'g'

  Selection: 'g'/'gl'/'gml' - 'g' means the velocity only has the group dependency, i.e., all angular moments are the same. 'gl' means the velocity also has the polar angle dependency. 'gml' means the velocity has the full angle dependency.

3.8.12 *DNPlambda*

Description: decay constants of delayed neutron precursors

Data type: a vector of real numbers

Default value: N/A

Unit: $1/s$

Note: the size must be equal to the number of delayed neutron precursor groups *I*. This cross section only valid for isotopes with classes of Fissile, Fertile, OtherActinides and Undefined in Table 3.

It has one attribute:

- *index*

  Description: subscript of the decay constant of delayed neutron precursors

  Data type: character(s)

  Default value: 'i'

  Selection: 'i' - 'i' means the delayed neutron fractions only have the delayed neutron group dependency.

4 A Sample Valid Library

```
<Multigroup_Cross_Section_Libraries Name="AP1000_A1" NGroup="7" >
<Multigroup_Cross_Section_Library ID="1" Ver="1.0"  Generator="Yaqi Wang"
   Description="3.1enr 12BP - pin diag adjacent to BP" >
  <!--
    Tabulation.
  -->
  <Tabulation>burnup temperature</Tabulation>
  <!--
    Grid points, library does not have to include all grid points.
  -->
  <burnup>0.0 1.0
  100</burnup>
  <temperature>300</temperature>
  <!--
    Reactions by default need mixing and interpolation. Check if these reactions are valid or not.
  -->
  <AllReactions>Total Scattering nuFission FissionSpectrum NeutronVelocity DNPlambda</AllReactions>
  <!--
    Table-wise reactions need no mixing but still need interpolation
  -->
  <TablewiseReactions>NeutronVelocity</TablewiseReactions>
  <!--
    Library-wise reactions need no mixing and interpolation
  -->
  <LibrarywiseReactions>DNPlambda</LibrarywiseReactions>
  <!--
    A single mixing table.
  -->
  <Table gridIndex="1 1">
    <Isotope Name="U235" MAT="9228" L="0" Class="Fissile" I="1" NS="1">
      <Total index="g">1.77949E-01 3.29805E-01 4.80388E-01 5.54367E-01 3.11801E-01 3.95168E-01 5.64406E-01</Total>
      <Scattering index="pgl" profile="1" has2l="false">
          <Profile>
            1 1
            1 2
            1 3
            1 5
            4 6
            5 7
            5 7
          </Profile>
          <Value>
             1.27537E-01
             4.23780E-02 3.24456E-01
             9.43740E-06 1.63140E-03 4.50940E-01
             5.51630E-09 3.14270E-09 2.67920E-03 4.52565E-01 1.25250E-04
             5.56640E-03 2.71401E-01 1.29680E-03
             1.02550E-02 2.65802E-01 8.54580E-03
             1.00210E-08 1.68090E-02 2.73080E-01
          </Value>
      </Scattering>
      <nuFission index="g">
        2.00600E-02 2.02730E-03 1.57060E-02 4.51830E-02 4.33421E-02 2.02090E-01 5.25711E-01
      </nuFission>
      <FissionSpectrum index="p">
        5.87910E-01 4.11760E-01 3.39060E-04 1.17610E-07 0.00000E+00 0.00000E+00 0.00000E+00
      </FissionSpectrum>
    </Isotope>
    <!--
      Table wise quantities.
    -->
    <Tablewise L="0" I="1" NS="1">
      <NeutronVelocity index="g">2e5 2e5 2e5 2e5 2e5 2e5 2e5</NeutronVelocity>
```

```
    </Tablewise>
  </Table>
  <!--
    Library wise quantities.
  -->
  <Librarywise L="0" I="1" NS="1">
    <DNPlambda index="i">1</DNPlambda>
  </Librarywise>
</Multigroup_Cross_Section_Library>
</Multigroup_Cross_Section_Libraries>
```

5 Cross Section Processing with YAKXS

5.1 Graphical view of YAKXS implementation

YAKXS contains ten c++ classes and one *Utility* sub-namespace:

1. *Utility* sub-namespace collects a set of useful data structures and functions for cross section processing, like converting a isotope name to its mat number, getting the default isotope class of an isotope, etc. Its contents can be found in the header file in Appendix A.
2. *MultigroupLibrary* holds the raw data loaded from a library in YAKXS format. It comes along with manipulators, accessors and writers.
3. *MixingTable* regulates the raw data for the mixing operation. It holds the microscopic cross sections at a particular state. *MixingTable* can be constructed from *MultigroupLibrary* through a selection or interpolation operation.
4. *Mixture* contains the macroscopic cross sections, which can be directly used by the transport solvers.
5. *MixedMultigroupLibrary* contains the macroscopic cross sections of all state points in *MultigroupLibrary*. *MixedMultigroupLibrary* can be obtained from *MultigroupLibrary* through mixing or folding operations. When folding operation is performed, new variables will be introduced and the atomic density dependency on these new variables is folded into the generated library.
6. *InputXS* is one of the macroscopic cross section holders. *InputXS* provides to transport solvers constant macroscopic cross sections.
7. *PerturbedInputXS* is one of the macroscopic cross section holders. *PerturbedInputXS* provides to transport solvers a simple model for cross sections with various feedback effects.
8. *FunctionInputXS* is one of the macroscopic cross section holders. Every cross section can be a function varying in space and time.
9. *YAKXSCreator* is designed for cross section generators to create the multigroup library in YAKXS format.
10. *TransmutationLibrary* holds the raw data loaded from a transmutation library.
11. *WorkingTransmutationLibrary* contains the preprocessed data from *TransmutationLibrary* which can be used for transmutation calculations directly. The number of isotopes in the *WorkingTransmutationLibrary* does not have to be the full list of isotopes in the *TransmutationLibrary*, from which the working library are generated.

All of them are included in *YAKXS namespace*. Cross section processing capabilities are built with the classes and utility functions. The relations among these classes are illustrated into Fig. 1.

Both *Mixture* and *InputXS* can be outputted into a data file in the INSTANT XML format. *Mixture* is not constructed from a data file in the INSTANT XML format. Both *MultigroupLibrary* and *MixedMultigroupLibrary* can be outputted into and be constructed from a data file in the YAKXS XML or binary format. Currently, *PerturbedInputXS* can only be constructed from a RELAP-5 input file containing the cross sections data with all the perturbation coefficients. We are planning to add the fitting capability so that we can convert a *MixedMultigroupLibrary* into *PerturbedInputXS*.

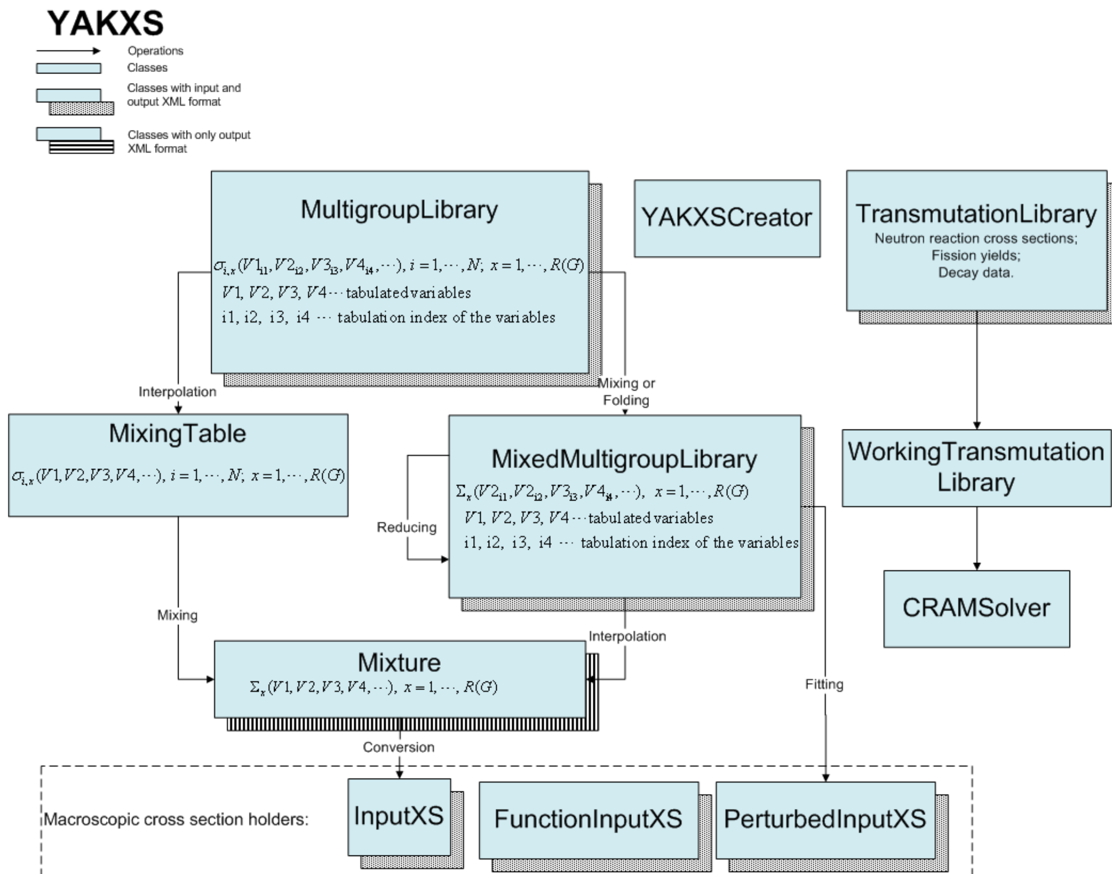

Figure 1 Graphical view of YAKXS.

5.2 Multigroup Library Creator

A c++ class *YAKXSCreator* is designed for creating the multigroup library in YAKXS format. Its interfaces can be found in Appendix A of the header file of YAKXSCreator.h. Users' task is to derive a concrete class from *YAKXSCreator* and to provide implementations of all the pure methods. Then the users can simply create an object with their derived class and call *createYAKXS()* to generate the library.

Syntax:

```
void createYAKXS(const std::string & fname, const std::string & libname) const;
```

Inputs:
fname - file name of the multigroup library in YAKXS format;
libname - name of the multigroup library.

5.3 Multigroup Library File Modifiers

Several modifiers of the library are provided under *YAKXS namespace* to directly modify the library in YAKXS format.

The modifier which inserts a reaction into a specified isotope:

```
int insertReaction(const std::string & reaction_name, const std::vector<double> & value,
                   const std::string & fname, const std::vector<unsigned int> &grid,
                   const std::string & isotope);
```

Inputs:
fname - file name of the multigroup library in YAKXS format;
grid - coordinates of state coordinates;
isotope - isotope name;
reaction_name - reaction name
value - cross sections

Returned value:
error code: 0 – the cross sections are inserted successfully.

Note: the default index of the reaction is assumed.

The modifier which inserts a isotope into a specified table:

```
int insertIsotope(const std::vector<std::string> & reaction_names,
                  const std::vector<std::vector<double> > & values,
                  const std::string & fname, const std::vector<unsigned int> &grid);
```

Inputs:
fname - file name of the multigroup library in YAKXS format;
grid - coordinates of state coordinates;
reaction_names - reaction names
values - cross sections

Returned value:
error code: 0 – the cross sections are inserted successfully.

Note: the default index of the reaction is assumed.

The modifier which inserts a table into the library:

```
int insertTable(const std::vector<std::vector<std::string> > & isotopes,
                const std::vector<std::vector<std::vector<double> > >& values,
                const std::string & fname, std::vector<unsigned int> &grid);
```

Inputs:
fname - file name of the multigroup library in YAKXS format;
grid - coordinates of the inserted table;
isotopes - a list of isotopes
values - cross sections

Returned value:
error code: 0 – the cross sections are inserted successfully.

The modifier which inserts a reaction into a table as a table-wise reaction:

```
int insertTablewise(const std::vector<std::string> & reaction_names,
                    const std::vector<std::vector<double> > & values,
                    const std::string & fname, std::vector<unsigned int> &grid);
```

The modifier which inserts a reaction into the library as a library-wise reaction:

```
int insertLibrarywise(const std::vector<std::string> & reaction_names,
                      const std::vector<std::vector<double> > & values,
                      const std::string & fname);
```

Meanings of arguments can be easily understood from the syntax.

It could be desired to have the modifier split into three steps: first loading the library then modifying the interior data and finally writing the data out. Although we already had the loader and writer, we do not have modifiers for the interior data provided.

5.4 Multigroup Library

A c++ class *MultigroupLibrary* is designed for manipulating the multigroup library in YAKXS format. Its interfaces can be found in the appendix of the short header file of MultigroupLibrary.h. All interfaces are cataloged into several classes which are described in the subsections.

5.4.1 The loader

The loader will load a library in some format. It will report any errors during loading. Currently only the YAKXS loader is functioning. ISOTXS and AMPX loaders are planed to be added soon.

Syntax:

```
int loadYAKXS(const std::string & fname, std::vector<std::string> & _msg);
```

Inputs:
fname - file name of the multigroup library in YAKXS format;

Outputs:
_msg - the error message generated by the loader indicating where the library is going wrong.

Returned value:
0 - means the library is loaded successfully;
1 - node not found;
2 - required attribute missing;

3 - unrecognized node;
4 - unrecognized attribute;
5 - duplicated node found;
6 - duplicated attribute found;
7 - invalid value.

Users can optionally call the following function to generate a screen printout about the error.

Syntax:

```
void reportErr(const int err, const std::vector<std::string> & _msg);
```

Its inputs are the output parameter and the returned value from the loader.

Right now the following three loaders are missing:

```
int loadISOTXS(const std::string &, std::vector<std::string> & ){ return 1; }
int loadAMPX(const std::string &, std::vector<std::string> &){ return 1; }
int loadNetCDF(const std::string &, std::vector<std::string> &){ return 1; }
```

5.4.2 The writer

The loaded library can be written out to a file in a format. Currently only the YAKXS writer is coded. Its syntax is as following:

```
void writeYAKXS(const std::string & fname, std::string libname = std::string());
```

Inputs:
fname - file name of the multigroup library in YAKXS format;
libname - [optional] the library name.

A potential binary writer could be:

```
void writeNetCDF(const std::string &, std::string) {}
```

5.4.3 The accessors

The following accessor for a loaded library to obtain a specific cross section is provided:

```
double getReaction(const std::vector<unsigned int> & grid_index,
                   const std::string & iso, const std::string & r,
                   unsigned int g = 1,
                   unsigned int gp= 1,
                   unsigned int l = 0,
                            int m = 0,
                   unsigned int i = 1,
                   unsigned int s = 1);
```

Inputs:
grid_index - coordinates of the inserted table;
isotope - isotope name;
r - reaction name;
g, gp, l, m, i, s - indices;

Returned value:
the cross section in double precision.

We also can get the library name through:

```
const std::string & name() const;
```

We can check if the unit of cross sections is in meter:

```
bool inMeter() const;
```

A list of isotopes for one of the tables can be obtained with

```
std::vector<std::string> getTableIsotopes(unsigned int tabid) const;
```

where the input is the table index.

There are also a set of table operations, which can be found in the header file. Their interfaces are self-explaining and will not be iterated here.

5.4.4 The extractors

An extractor is defined as the subroutine to create a subset of the data through operations. For example, a single table for mixing purpose can be extracted from the multigroup library either through interpolation or simply through selecting one contained table. Or a multigroup macroscopic cross section library can be built through the mixing or folding operation. By this definition, an extractor is also a constructor of another YAKXS class, like the mixing table and the mixed multigroup library. An explanation will be included in the sections of the extracted classes.

5.5 Mixing Table

The mixing table is closely related with the multigroup library. A mixing table contains all the necessary cross sections at a state point for the mixing. These cross sections are organized in a way that the mixing can be done efficiently. The detailed interface for the c++ class MixingTable can be found in the appendix. Right now there are two ways of construction as mentioned in Section 5.4.4: interpolation or selection.

The syntax for selection is:

```
MixingTable(MultigroupLibrary & lib, std::vector<unsigned int> & grid);
```

where the inputs are the multigroup library *lib* and the coordinate of the selected table *grid*.

The syntax for interpolation is:

```
MixingTable(MultigroupLibrary & lib, std::vector<double> & grid_values){}
```

where the inputs are the multigroup library *lib* and the coordinate value of state variables *grid_values*. Right now, we have to have the full table for the interpolation.

5.6 Mixture

MixingTable is used only by mixtures which also hold the isotope densities, which explained the syntax of the constructor of a mixture

```
Mixture(MixingTable & mixing_table, std::map<std::string, double> densities, bool enable_update=true);
```

The third argument indicates that if we want to update the mixture with changed densities. If it is true, we can call the following two overloaded functions and another erasing function can perform this task.

```
void setDensity(std::string isotope, double density);
void setDensity(std::map<std::string, double> densities);
void eraseIsotope(const std::string & isotope);
```

Whenever the densities of isotopes are changed, the mixing operation need to be done once again. This operation are hidden in the above three functions and the constructor. To be able to do this, the MixingTable must stay valid. One the other hand, if we know the mixture will stay constant after its construction, we can delete MixingTable used for the construction.

The mixture provides a bunch of accessors to get the macroscopic cross section out, which are self-explained:

```
  inline unsigned int getNumberOfGroups() const;
  inline unsigned int getNumberOfDelayGroups() const;
  inline unsigned int getNA() const;
  inline bool hasNeutronSpeed() const;
  inline bool isFissile() const;
  inline bool assertNotVacuum() const;
  inline bool hasDiffusion() const;
  std::vector<unsigned int> scattering_profiling() const;
  double getReaction(ReactionType r,
                     unsigned int g = 1,
                     unsigned int gp= 1,
                     unsigned int l = 0,
                              int m = 0,
                     unsigned int i = 1,
                     unsigned int s = 1) const;
  double getTotal(unsigned int g, unsigned int l=0, int m=0) const;
  double getAbsorption(unsigned int g) const;
  double getScattering(unsigned int gp, unsigned int g,
                       unsigned int l=0, int m=0) const;
  double getNuFission(unsigned int gp) const;
  double getFission(unsigned int g) const;
  double getKappaFission(unsigned int g) const;
  double getFissionSpectrum(unsigned int g, unsigned int gp=1) const;
  double getRemoval(unsigned int g) const;
  double getTransport(unsigned int g, unsigned int l=0, int m=0) const;
  double getNeutronVelocity(unsigned int g, unsigned int l=0, int m=0) const;
  double getDNPlambda(unsigned int i) const;
  double getDNFraction(unsigned int i, unsigned int gp=1) const;
  double getDNSpectrum(unsigned int i, unsigned int gp) const;
```

A mixture can also write out the macroscopic cross sections directly for transport calculation. To this purpose, a subroutine

```
void writeINSTANTMaterials(const std::string & fname, const unsigned int id=1) const;
```

is provided to generate the XML material block for INSTANT. *id* in the argument list is the optional material ID. Refer to INSTANT XML format in the appendix of this document for more details.

5.7 Mixed Multigroup Library

A mixed multigroup library can be constructed in three ways. First we can read a pseudo isotope in the YAKXS library:

```
MixedMultigroupLibrary(const std::string & fname, const std::string & macro);
```

where macro is the pseudo isotope name.

Or it can be constructed through the mixing operation from a multigroup library by giving a list of atomic densities.

```
MixedMultigroupLibrary(const MultigroupLibrary & lib, const std::map<std::string, double> densities);
```

Finally it can be constructed through the folding operation from a multigroup library.

```
MixedMultigroupLibrary(const MultigroupLibrary & lib,
                       const std::vector<std::string> & grid_names,
                       const std::map<std::vector<double>, std::map<std::string, double> >& densities);
```

A new set of grids will be introduced through this operation. The number of new grids is equal to the size of the second argument *grid_names*. The folding operation is currently missing.

A mixed multigroup library can be written into a file in YAKXS XML format with

```
void writeYAKXS(const std::string & fname, std::string libname = std::string(), std::string macro = "Macro") const;
```

where we can give a new library name in *libname* and a new psedo isotope name in *macro*.

We can get cross sections through

```
 double getReaction(const std::map<std::string, unsigned int> & grid_index,
                    ReactionType r,
                    unsigned int g = 1,
                    unsigned int gp= 1,
                    unsigned int l = 0,
                             int m = 0,
                    unsigned int i = 1,
                    unsigned int s = 1) const;
```

or get a mixture with

```
const Mixture & getMixture(unsigned int tabid) const;
```

where *tabid* is the table index. We can get the library name with

```
const std::string & name() const;
```

All table operations valid in *MultigroupLibrary* are valid for *MixedMultigroupLibrary*.

The length unit of all cross sections can be converted to meter or centimeter through

```
void convert2meter();
void convert2centimeter();
```

And the current length unit can be inquired with

```
bool inMeter() const;
```

5.8 Macroscopic Cross Section Holders

As the name suggested, macroscopic cross section holders hold macroscopic cross sections. There are currently three types of macroscopic cross section holders. One holds the macroscopic cross sections for a material at one single state point. The class corresponding to this holder is *InputXS*, whose header file can be found in Appendix A. The second one holds macroscopic cross sections at a reference state point as well as the perturbation parameters of a selected subset of state variables. The class corresponding to this holder is *PerturbedInputXS*, whose header file can be found in Appendix A. The last one is mainly for benchmarking purpose. Users can give the macroscopic cross sections as functions varying in space and time.

5.8.1 InputXS

There are essentially three ways of constructing *InputXS*: a sequential constructing function calls, loading a data file in INSTANT XML format, which is detailed in Appendix D, or to be converted from a YAKXS mixture with a fixed set of isotope densities.

In the first way of construction, we perform the following steps:

1. Set total and scattering cross section. If diffusion coefficient, removal cross section or absorption is empty, it will be derived with the following equations:

$$D_g = \frac{1}{3(\sigma_{t,g} - \sigma_{s,1}^{g\to g})}$$
$$\sigma_{r,g} = \sigma_{t,g} - \sigma_{s,0}^{g\to g}$$
$$\sigma_{a,g} = \sigma_{t,g} - \sum_{g'=1}^{G} \sigma_{s,0}^{g\to g'}.$$

2. Set ν fission cross section and fission spectrum for fissile materials. If ν (averaged neutron emission per fission) or κ (averaged recoverable energy released per fission) is empty, it will be set to a default value 2.43 and 195MeV. Fission cross section and κ fission cross section will be then derived.
3. Optionally set fission cross section and calculate ν from it for fissile materials.
4. Optionally Set κ fission cross section and calculate kappa from it for fissile materials.
5. Set delayed neutron data if the number of delayed neutron groups is greater than zero.
6. Set averaged neutron speeds for transient calculations.
7. Set diffusion coefficient, removal cross section and scattering cross section. If total cross section is empty, it will be derived and in-group scattering cross section will be derived as well with the following equation:

$$\sigma_{t,g} = \frac{1}{3D_g}$$
$$\sigma_{s,0}^{g\to g} = \sigma_{t,g} - \sigma_{r,g}.$$

In the second way, we simply call

```
unsigned int loadINSTANTXS(const std::string & fname, const unsigned int mid,
                           std::vector<std::string> & msg);
```

where,
fname - the file name of the data file in INSTANT XML format;
mid - material ID;

msg - error message generated when loading.
When error happens during the loading, the function returns a nonzero integer. Otherwise it returns zero.

In the third way, we simply call

```
void buildFromMixture(const YAKXS::Mixture & mix);
```

where,
mix - the YAKXS mixture.

Once *InputXS* is constructed, we can assert the material not to be vacuum by calling

```
  void assertNotVacuum() const;
```

and we can write the data into a plain text file with

```
  void write(const std::string & fname) const;
```

or write into a INSTANT XML file with

```
  void writeINSTANTXS(const std::string & fname, const unsigned int mid) const;
```

We can also obtain the profiling of the scattering matrices with

```
  std::vector<unsigned int> scattering_profiling() const;
```

Profiling is described in detail in Appendix D.

There are also few manipulators.

```
  void transposeForAdjoint();
```

transposes scattering matrices for the ajoint calculations.

```
  void convert2meter();
```

converts data into unit of meter assuming originally in unit of centimeter.

Some simple accessors to get the cross section out can be found in Appendix A.

5.8.2 PerturbedInputXS

There is only one way of construction through reading a plain data file extracted and modified from RELAP5 input. We can convert a YAKXS at a given set of isotope densities and a given state point with fitted perturbation parameters on selected variables to this cross section holder in the future. The class corresponding to this holder is *PerturbedInputXS*, whose header file can be found in Appendix A. Any perturbed macroscopic cross sections denoted with x is evaluated with

$$\sigma_x = \sigma_{x,0} + \sum_{i=1}^{N} \sum_{j=1}^{M_i} \gamma_{x,i,j} (v_i - v_{i,0})^j, \tag{1}$$

where N is the number of perturbation variables and M_i is the order of the perturbation for a given variable v_i. $\sigma_{x,0}$ is the cross section at the reference state and $v_{i,0}$ is the reference value of the $i-$th variable. Alternatively, cross sections can be evaluated with

$$\sigma_x = \sigma_{x,0} + \sum_{i=1}^{N} \sum_{j=1}^{M} \gamma_{x,i,j} (\sqrt{v_i} - \sqrt{v_{i,0}})^j, \tag{2}$$

where variable values are squre rooted.

Interface of the construction of *PerturbedInputXS* is

```
void Relap5Reader(unsigned int ngroup, const std::string & filename, unsigned int material_id,
                  unsigned int submat_id,
                  bool dbg);
```

where,
ngroup - number of energy groups filename - the file name of the plain text data file;
material_id - material ID;
submat_id - 1/2/3 – rodded/unrodded/driver;
dbg - true to show debugging messages on screen while loading the file.

It has some methods like *InputXS*:

```
void transposeForAdjoint();
void convert2meter();
void assertNotVacuum() const;
void write(const std::string & fname, const unsigned int id=0) const;
```

It can return number of perturbation variables and their reference values:

```
inline unsigned int getNParam() const;
inline const std::vector<Real>& getReferenceParams() const;
```

All other cross sections accessors now accept the values of perturbation variables. If the values of perturbation variables are missing, these accessors return the cross sections at the reference state point.

5.8.3 FunctionInputXS

There is only one way for constructing *FuncitonInputXS*: a sequential constructing function calls. The same steps are followed during the construction as those of the first way of constructing *InputXS*.

Once *FuncitionInputXS* is constructed, we can assert the material not to be vacuum by calling

```
void assertNotVacuum() const;
```

We can transposes scattering matrices for the adjoint calculations with

```
void transposeForAdjoint();
```

The sparsity of the scattering matrix can be inquired at a list of sampling temporal and spatial points by

```
  bool checkNonzeroScattering(unsigned int gp, unsigned int g) const;
```

Note that the sampling points are the inputs when the object is created. Similarly we can check the fission spectrum by

```
bool checkNonzeroSpectrum(unsigned int g) const;
```

Some simple accessors to get the cross section out can be found in Appendix A.

5.9 Transmutation Library

5.10 Working Transmutation Library

Appendices

A Header Files

To avoid distraction, all private members of classes are removed from the original header file. The *YAKXS namespace* is also removed from the original header file.

1. YAKXSUtilities.h
   Note: general utility functions are removed from the header.

```
// MT numbers
enum ReactionType
{
  Total = 1,
  Absorption = 27,
  Fission = 18,
  Removal = -1,
  Transport = -2,
  Scattering = -3,
  nuFission = -4,
  kappaFission = -5,
  FissionSpectrum = -6,
  DNFraction = -7,
  DNSpectrum = -8,
  NeutronVelocity = -9,
  DNPlambda = -10,
  InvalidReactionType = -999
};
// Isotope classes
enum IsotopeClass
{
  InvalidIsotopeClass = -1,
  Undefined = 0,
  Fissile = 1,
  Fertile = 2,
  OtherActinide = 3,
  FissionProduct = 4,
  Structure = 5,
  Coolant = 6,
  Control = 7
};
enum MixingType
{
  Weighted_Summation, // for normal cross sections
  Weighted_Average,   // for quantities like the fission spectrum
  No_Weighting,       // for quantities like the neutron velocity
  InvalidMixingType
};
namespace Utility {
/* yakxs utilities */
template <typename T>
```

```
T string_to_enum (const std::string& s);
template <typename T>
std::string enum_to_string (const T e);
int string_to_mat(const std::string& s);
std::string mat_to_string(const int mat);
int helios_id_to_mat(int id);
IsotopeClass mat_to_class(const int mat);
bool is_class_has_reaction(const IsotopeClass iclass, const ReactionType rtype);
bool isValidMAT(const int mat);
MixingType reaction_mixing_type(ReactionType r);
std::string getDefaultIndex(ReactionType rtype);
int checkIndexConsistency(ReactionType, const std::string & ind);
bool isPseudoIsotope(const std::string & name);
std::string getElementName(unsigned int Z);
std::string getElementNameFromIsotopeName(const std::string & isot);
bool isElementNameValid(const std::string & isot);
void getAZFromIsotopeName(const std::string & s, unsigned int & A, unsigned int & Z);
/* the grid */
class Grid
{
public:
  /* accessors */
  std::vector<std::string> & setGridNames() { return _gridnames; }
  std::map<std::string, std::vector<double> > & setGrids() { return _grids; }
  std::map<std::string, std::vector<double> > & setGridsTol() { return _grids_tol; }
  std::vector<std::vector<unsigned int> > & setGridIndices() { return _grid_indices; }
  std::map<std::string, unsigned int> & setReferenceGrid() { return _ref_grids; }
  const std::vector<std::string> & getGridNames() const { return _gridnames; }
  const std::map<std::string, std::vector<double> > & getGrids() const { return _grids; }
  const std::map<std::string, std::vector<double> > & getGridsTol() const { return _grids_tol; }
  const std::vector<std::vector<unsigned int> > & getGridIndices() const { return _grid_indices; }
  const std::map<std::string, unsigned int> & getReferenceGrid() const { return _ref_grids; }
  /* interpolation utilities */
  unsigned int find_table(const std::map<std::string, unsigned int> & grid_index) const;
  void find_tables(const std::map<std::string, double> & grid_values,
                   std::vector<unsigned int> & tab_index, std::vector<double> & tab_weights) const;
  bool hasFullTable() const;
  bool isGridPivoting(const std::string & grid_name) const;
  std::string interpol_pivot() const;
  /* manipulators */
  void reduceDimensionality(const std::set<std::string> & grids);
  /* simple utilities */
  const std::vector<double> & getGridValues(const std::string & grid_name) const;
  std::map<std::string, double> getReferenceGridValues() const;
  unsigned int getReferenceTable() const;
  std::map<std::string, double> getTableGridValues(unsigned int tabid) const;
  std::map<std::string, unsigned int> getTableGrids(unsigned int tabid) const;
  std::vector<unsigned int> getTablesOnDimension(const std::string & grid) const;
  std::vector<unsigned int> getTablesOffDimension() const;
};
}
/* end of Utility*/
int insertReaction(const std::string & reaction_name, const std::vector<double> & value,
                   const std::string & fname, const std::vector<unsigned int> &grid,
                   const std::string & isotope);
int insertIsotope(const std::vector<std::string> & reaction_names,
                  const std::vector<std::vector<double> > & values,
                  const std::string & fname, const std::vector<unsigned int> &grid);
int insertTable(const std::vector<std::vector<std::string> > & isotopes,
                const std::vector<std::vector<std::vector<double> > >& values,
                const std::string & fname, std::vector<unsigned int> &grid);
int insertTablewise(const std::vector<std::string> & reaction_names,
                    const std::vector<std::vector<double> > & values,
                    const std::string & fname, std::vector<unsigned int> &grid);
int insertLibrarywise(const std::vector<std::string> & reaction_names,
                      const std::vector<std::vector<double> > & values,
                      const std::string & fname);
}
```

2. YAKXSCreator.h

```
class YAKXSCreator
{
public:
```

```
  /*
   * Create YAKXS library with all call-back functions provided by the inherited classes
   */
  virtual void createYAKXS(const std::string & fname, const std::string & libname, unsigned int libid) const;
protected:
  /*
   * Comment about the library.
   * Note: this comment will go into the XML file.
   */
  virtual std::string libraryComment() const;
  /*
   * Return the number of energy groups
   */
  virtual unsigned int numberOfEnergyGroups() const = 0;
  /*
   * Return the number of groups of delay neutron precursors
   * Note: return 0 if no delay neutron data present.
   * Note: Calling with default tableID will return library-wise value.
   *       Calling with default isotope will return table-wise value.
   */
  virtual unsigned int numberOfDelayNeutronGroups(unsigned int tableID=Utility::InvalidTableID,
                                                  std::string isotope="Tablewise") const = 0;
  /*
   * Return the name of the creator
   * Note: Default value is 'INL'.
   */
  virtual std::string nameOfCreator() const;
  /*
   * Return true if cross section data is in unit of meter.
   */
  virtual bool inMeter() const;
  /*
   * Return a list of valid reaction types in the library
   */
  virtual std::vector<ReactionType> allReactionTypes() const = 0;
  /*
   * Return names of grids (coordinates) for the tabulated cross sections
   */
  virtual std::vector<std::string> nameOfGrids() const = 0;
  /*
   * Return the reference index of all grids
   */
  virtual std::map<std::string, unsigned int> referenceGridIndex() const;
  /*
   * Return unit of a specific grid
   */
  virtual std::string gridUnit(const std::string & grid_name) const;
  /*
   * Return values of a specific grid
   */
  virtual std::vector<double> gridValues(const std::string & grid_name) const = 0;
  /*
   * Return the number of tables
   * Note: the number of tables does not have to be equal to the product of number of values of all grid.
   */
  virtual unsigned int numberOfTables() const = 0;
  /*
   * Return indices of all grids for a given table
   */
  virtual std::vector<unsigned int> gridIndices(unsigned int tableID) const = 0;
  /*
   * Return the list of names of isotopes in the table
   */
  virtual std::vector<std::string> isotopesOfTable(unsigned int tableID) const = 0;
  /*
   * Return isotope MAT number
   */
  virtual int isotopeMat(std::string isotope="Tablewise") const;
  /*
   * Return the list of reaction types
   * Note: Calling with default tableID will return library-wise reaction types.
   *       Calling with default isotope will return table-wise reaction types.
   */
  virtual std::vector<ReactionType> reactionTypes(unsigned int tableID=Utility::InvalidTableID,
                                                  std::string isotope="Tablewise") const = 0;
  /*
   * Return the maximum order of spherical harmonics (L)
   * Note: Calling with default tableID will return the library-wise value.
```

```
     *       Calling with default isotope will return the table-wise value.
     */
    virtual unsigned int orderOfSphericalHarmonics(unsigned int tableID=Utility::InvalidTableID,
                                               std::string isotope="Tablewise") const = 0;
    /*
     * Return the number of shape function for spatial expansion (NS)
     * Note: Calling with default tableID will return library-wise reaction types.
     *       Calling with default isotope will return table-wise reaction types.
     *       Default to 0 because this is a feature not always being used.
     */
    virtual unsigned int numberOfSpatialMoments(unsigned int tableID=Utility::InvalidTableID,
                                            std::string isotope="Tablewise") const;
    /*
     * True to indicate that the scattering cross section has the 2l+1 coefficient
     * Note: defaul to false because it should become a standard later on.
     */
    virtual bool has2LScattering(unsigned int tableID=Utility::InvalidTableID,
                             std::string isotope="Tablewise") const;
    /*
     * Return the scattering profile option
     * Note: Calling with default tableID will work for a library-wise reaction.
     *       Calling with default isotope will work for a table-wise reaction.
     */
    virtual unsigned int profileOfScattering(unsigned int tableID=Utility::InvalidTableID,
                                         std::string isotope="Tablewise") const = 0;
    /*
     * Return the upper profile bound
     * Note: Calling with default tableID will work for a library-wise reaction.
     *       Calling with default isotope will work for a table-wise reaction.
     *       refer to YAKXS mannul for further explanation of profiling.
     *       Profile is only applied for scattering cross sections currently.
     */
    virtual std::vector<unsigned int> upperProfileBound(unsigned int tableID=Utility::InvalidTableID,
                                                    std::string isotope="Tablewise") const = 0;
    /*
     * Return the lower profile bound
     * Note: Calling with default tableID will work for a library-wise reaction.
     *       Calling with default isotope will work for a table-wise reaction.
     *       refer to YAKXS mannul for further explanation of profiling.
     *       Profile is only applied for scattering cross sections currently.
     */
    virtual std::vector<unsigned int> lowerProfileBound(unsigned int tableID=Utility::InvalidTableID,
                                                    std::string isotope="Tablewise") const = 0;
    /*
     * Return the index of the cross sections of a reaction
     * Note: Calling with default tableID will work for a library-wise reaction.
     *       Calling with default isotope will work for a table-wise reaction.
     *       Do not modify this if a reaction index is fixed in the YAKXS format.
     */
    virtual std::string indexOfReaction(ReactionType rt,
                                    unsigned int tableID=Utility::InvalidTableID,
                                    std::string isotope="Tablewise") const;
    /*
     * Return the cross section
     * Note: Calling with default tableID will work for a library-wise reaction.
     *       Calling with default isotope will work for a table-wise reaction.
     */
    virtual std::vector<double> crossSectionsOfReaction(ReactionType rt,
                                                    unsigned int tableID=Utility::InvalidTableID,
                                                    std::string isotope="Tablewise") const = 0;
};
```

3. MultigroupLibrary.h

```
class MultigroupLibrary
{
public:
  int loadYAKXS(const std::string & fname, const std::string & libname, unsigned int MatID, std::vector<std::str
  void reportErr(const int err, const std::vector<std::string> & _msg) const;
  int loadISOTXS(const std::string &, std::vector<std::string> & ){ return 1; }
  int loadAMPX(const std::string &, std::vector<std::string> &){ return 1; }
  int loadNetCDF(const std::string &, std::vector<std::string> &){ return 1; }
  void writeYAKXS(const std::string & fname, std::string libname, unsigned int material_id) const;
  void writeNetCDF(const std::string &, std::string) const {}
  void convert2meter();
  void convert2centimeter();
 /* data access */
```

```
  double getReaction(const std::map<std::string, unsigned int> & grid_index,
                     const std::string & iso, const std::string & r,
                     unsigned int g = 1,
                     unsigned int gp= 1,
                     unsigned int l = 0,
                              int m = 0,
                     unsigned int i = 1,
                     unsigned int s = 1) const;
  const std::string & name() const { return _name; }
  bool inMeter() const { return _inmeter; }
  std::vector<std::string> getTableIsotopes(unsigned int tabid) const;
  unsigned int getNumberOfEnergyGroups() const { return _ngroup; }
  bool isReactionConstant(ReactionType r) const;
  unsigned int getMaximumNA() const;
  bool hasReaction(ReactionType r) const;
  std::vector<bool> fissionPattern() const;
  std::vector<std::vector<bool> > scatteringPattern() const;
  /* grid utilities */
  unsigned int find_table(const std::map<std::string, unsigned int> & grid_index) const;
  void find_tables(const std::map<std::string, double> & grid_values,
                   std::vector<unsigned int> & tab_index, std::vector<double> & tab_weights) const;
  bool hasFullTable() const;
  bool isGridPivoting(const std::string & grid_name) const;
  std::string interpol_pivot() const;
  const std::vector<std::string> & getGridNames() const;
  const std::string getGridUnit(const std::string & grid_name) const;
  const std::vector<double> & getGridValues(const std::string & grid_name) const;
  const std::map<std::string, std::vector<double> > & getGridMap() const;
  std::map<std::string, unsigned int> referenceGrid() const;
  std::map<std::string, double> referenceGridValues() const;
  unsigned int referenceTable() const;
};
/* end of MultigroupLibrary */
```

4. MixingTable.h

```
class MixingTable
{
public:
  /* construct from a cross section file  */
  MixingTable(const MultigroupLibrary & lib, const std::map<std::string, unsigned int> & grid);
  /* construct from an interpolation table with interpolation parameters  */
  MixingTable(const MultigroupLibrary & lib, const std::map<std::string, double> & grid_values);
};
/* end of MixingTable */
```

5. Mixture.h

```
class Mixture
{
public:
  Mixture(MixingTable & mixing_table, std::map<std::string, double> densities, bool enable_update=true);
  Mixture(const MixedMultigroupLibrary & lib, const std::map<std::string, unsigned int> & grid);
  Mixture(const MixedMultigroupLibrary & lib, const std::map<std::string, double> & grid_values);
  inline unsigned int getNumberOfGroups() const
    {
      return _reaction_index_limit[0].ngroup;
    }
  inline unsigned int getNumberOfDelayGroups() const
    {
      if (_dnfraction_id == _reaction_index_limit.size())
        return 0;
      else
        return _reaction_index_limit[_dnfraction_id].ndnp;
    }
  inline unsigned int getNA() const
    {
      return _reaction_index_limit[_scattering_id].na;
    }
  inline bool hasNeutronSpeed() const
    {
      return (_neutronvelocity_id != _reactions.size());
    }
  inline bool isFissile() const
    {
```

```
      return (_fissionspectrum_id != _reactions.size());
    }
  inline bool assertNotVacuum() const
    {
      bool flag = true;
      for (unsigned int i=0; i<getNumberOfGroups(); i++)
        if (getTotal(1) == 0.0) flag = false;
      return flag;
    }
  inline bool hasDiffusion() const
    {
      return ((_transport_id != _reactions.size()) && (_removal_id != _reactions.size()));
    }
  std::vector<unsigned int> scattering_profiling() const
    {
      std::vector<unsigned int> p;
      for (unsigned int n=0; n<=getNA(); n++)
        for (unsigned int g=0; g<getNumberOfGroups(); g++)
        {
          int gp=0;
          for (; gp<int(getNumberOfGroups()); gp++)
            if (getScattering(gp+1, g+1, n) != 0.0) break;
          gp++;
          p.push_back(gp);
          gp=getNumberOfGroups()-1;
          for (; gp>=0; gp--)
            if (getScattering(gp+1, g+1, n) != 0.0) break;
          gp++;
          p.push_back(gp);
        }
      return p;
    }
  double getReaction(ReactionType r,
                     unsigned int g = 1,
                     unsigned int gp= 1,
                     unsigned int l = 0,
                              int m = 0,
                     unsigned int i = 1,
                     unsigned int s = 1) const;
  double getTotal(unsigned int g, unsigned int l=0, int m=0) const;
  double getAbsorption(unsigned int g) const;
  double getScattering(unsigned int gp, unsigned int g,
                       unsigned int l=0, int m=0) const;
  double getNuFission(unsigned int gp) const;
  double getFission(unsigned int g) const;
  double getKappaFission(unsigned int g) const;
  double getFissionSpectrum(unsigned int g, unsigned int gp=1) const;
  double getRemoval(unsigned int g) const;
  double getTransport(unsigned int g, unsigned int l=0, int m=0) const;
  double getNeutronVelocity(unsigned int g, unsigned int l=0, int m=0) const;
  double getDNPlambda(unsigned int i) const;
  double getDNFraction(unsigned int i, unsigned int gp=1) const;
  double getDNSpectrum(unsigned int i, unsigned int gp) const;
  void setDensity(std::string isotope, double density);
  void setDensity(std::map<std::string, double> densities);
  void eraseIsotope(const std::string & isotope);
  void writeINSTANTMaterials(const std::string & fname, unsigned int id=1) const;
};
/* end of Mixture */
```

6. MixedMultigroupLibrary.h

```
class MixedMultigroupLibrary
{
public:
  /**
   * reading constructor
   */
  MixedMultigroupLibrary(const std::string & fname, const std::string & libname, unsigned int MatID, const std:
  /**
   * mixing constructor
   */
  MixedMultigroupLibrary(const MultigroupLibrary & lib, const std::map<std::string, double> densities);
  /**
   * folding constructor
   */
```

```
  MixedMultigroupLibrary(const MultigroupLibrary & lib,
                         const std::vector<std::string> & grid_name,
                         const std::map<std::vector<double>, std::map<std::string, double> >& densities);
  /**
   * reducing constructor
   */
  MixedMultigroupLibrary(const MixedMultigroupLibrary & mlib,
                         const std::map<std::string, double> & grid_values);
  void writeYAKXS(const std::string & fname, const std::string & libname = "lib_data", unsigned int MatID = 1, s
  void writeNetCDF(const std::string &, std::string) const {}
  void convert2meter();
  void convert2centimeter();
  bool isReactionConstant(ReactionType r) const;
  bool hasReaction(ReactionType r) const { return _mixtures[0]->hasReaction(r); }
  double getReaction(const std::map<std::string, unsigned int> & grid_index,
                     ReactionType r,
                     unsigned int g = 1,
                     unsigned int gp= 1,
                     unsigned int l = 0,
                              int m = 0,
                     unsigned int i = 1,
                     unsigned int s = 1) const;
  const std::string & name() const { return _name; }
  bool inMeter() const { return _inmeter; }
  const Mixture & getMixture(unsigned int tabid) const { return *_mixtures[tabid]; }
  unsigned int getNumberOfEnergyGroups() const { return _ngroup; }
  bool isFissile() const { return _mixtures[0]->isFissile(); }
  unsigned int getNA() const { return _mixtures[0]->getNA(); }
  unsigned int getNumberOfDelayGroups() const { return _mixtures[0]->getNumberOfDelayGroups(); }
  unsigned int find_table(const std::map<std::string, unsigned int> & grid_index) const;
  void find_tables(const std::map<std::string, double> & grid_values,
                   std::vector<unsigned int> & tab_index, std::vector<double> & tab_weights) const;
  const std::vector<std::string> & getGridNames() const;
  const std::string getGridUnit(const std::string & grid_name) const;
  const std::vector<double> & getGridValues(const std::string & grid_name) const;
  const std::map<std::string, std::vector<double> > & getGridMap() const;
  std::map<std::string, unsigned int> referenceGrid() const;
  std::map<std::string, double> referenceGridValues() const;
  unsigned int referenceTable() const;
  std::map<std::string, double> getTableGridValues(unsigned int tabid) const
    { return _grid.getTableGridValues(tabid); }
  std::vector<unsigned int> getTablesOnDimension(const std::string & grid) const
    { return _grid.getTablesOnDimension(grid); }
  std::vector<unsigned int> getTablesOffDimension() const { return _grid.getTablesOffDimension(); }
  bool hasFullTable() const;
  bool isGridPivoting(const std::string & grid_name) const;
  std::string interpol_pivot() const;
};
/* end of MixedMultigroupLibrary */
```

7. InputXS.h

```
class InputXS
{
public:
  InputXS();
  InputXS(const std::string & name);
  /* Followin 7 methods provide a flexiable way of constructing InputXS */
  /*
   * Set total and scattering cross section, if diffusion coefficient, removal cross section
   * or absorption is empty, it will be derived.
   */
  void setTotalAndScattering(unsigned int ngroup,
                             std::vector<double> sigt,
                             unsigned int NA,
                             std::vector<double> sigs);
  /*
   * Set nu fission cross section and fission spectrum which are necessary for criticality calculations.
   * If nu or kappa is empty, it will be set a default value 2.43 and 195MeV. Fission cross section and
   * kappa fission cross section will be then derived.
   */
  void setFission(unsigned int ngroup,
                  std::vector<double> nusigf,
                  std::vector<double> chi);
```

```
  /*
   * Set fission cross section and calculate nu from it and the nu fission cross section.
   * Note: this method has to be called after setFission() if setFission() is to be called.
   */
  void setFission1(unsigned int ngroup,
                   std::vector<double> sigf);
  /*
   * Set kappa fission cross section and calculate kappa from it and the fission cross section.
   * Note: this method has to be called after setFission1() if setFission1() is to be called.
   */
  void setFission2(unsigned int ngroup,
                   std::vector<double> ksigf);
  /*
   * Set capture cross section.
   */
  void setCapture(unsigned int ngroup,
                  std::vector<double> sig_capture);
   /*
   * Set nalpha cross section.
   */
  void setNalpha(unsigned int ngroup,
                 std::vector<double> sig_nalpha);
  /*
   * Set delay neutron data.
   */
  void setDelayNeutron(unsigned int ngroup,
                       unsigned int n_delay_groups,
                       std::vector<double> decay_constant,
                       std::vector<double> delay_neutron_fraction,
                       std::vector<double> delay_chi);
  /*
   * Set neutron speeds.
   */
  void setNeutronSpeed(unsigned int ngroup,
                       std::vector<double> neutron_speed);
  /*
   * Set diffusion coefficient, removal cross section and scattering cross section.
   * If total cross section is empty, it will be derived and in-group scattering cross section will
   * be derived as well.
   */
  void setDiffusionRemoval(unsigned int ngroup,
                           std::vector<double> diffusion_coeff,
                           std::vector<double> sigma_r,
                           std::vector<double> sigma_s);
  /*
   * Construct InputXS from a data file.
   */
  void setTotalScatteringAndFission(const std::string & fname,
                                    unsigned int materal_num,
                                    unsigned int ngroups,
                                    bool dbg=false);
#ifdef RAPIDXML_HPP_INCLUDED
  /*
   * Construct InputXS from a INSTANT XML file.
   * Note: rapidXML is required by this method!
   */
  unsigned int loadINSTANTXS(const std::string & fname, const unsigned int mid,
                             std::vector<std::string> & msg);
  /*
   * Read one XML material from the node.
   */
  int
  readXMLMaterial(xml_node<> *child, unsigned int id, unsigned int ngroup, unsigned int ndnp,
                  unsigned int na, bool fis, std::vector<std::string> & msg);
#endif
  virtual void transposeForAdjoint();
  virtual void convert2meter();
  virtual void convert2centimeter();
  virtual void assertNotVacuum() const;
  virtual void output(std::ostream & os) const;
  std::vector<unsigned int> scattering_profiling() const;
  /*
   * Simple accessors.
   */
  inline double getSigmaTotal(unsigned int g) const
    {
      return _sigma_t[g];
    }
  inline double getSigmaAbsorption(unsigned int g) const
```

```
    {
      return _sigma_a[g];
    }
  inline double getSigmaCapture(unsigned int g) const
    {
      return _sigma_capture[g];
    }
  inline double getSigmaNalpha(unsigned int g) const
    {
      return _sigma_nalpha[g];
    }
  inline double getSigmaRemoval(unsigned int g) const
    {
      return _sigma_r[g];
    }
  inline double getDiffusionCoefficient(unsigned int g) const
    {
      return _diffusion_coeff[g];
    }
  inline double getNuSigmaFission(unsigned int g) const
    {
      return _nu_sigma_f[g];
    }
  inline double getSigmaFission(unsigned int g) const
    {
      return _sigma_f[g];
    }
  inline double getKappaSigmaFission(unsigned int g) const
    {
      return _kappa_sigma_f[g];
    }
  inline double getNu(unsigned int g) const
    {
      return _nu[g];
    }
  inline double getKappa(unsigned int g) const
    {
      return _kappa[g];
    }
  inline double getFissionSpectrum(unsigned int g) const
    {
      return _chi[g];
    }
  inline double getSigmaScattering(unsigned int gp, unsigned int g, unsigned int n) const
    {
      if (n>getNA()) return 0;
      else
return _sigma_s[n*_ngroup*_ngroup + g*_ngroup + gp];
    }
  inline double getSigmaScattering(unsigned int gp, unsigned int g) const
    {
      return _sigma_s[g*_ngroup + gp];
    }
  inline double getNeutronSpeed(unsigned int g) const
    {
      return _neutron_speed[g];
    }
  inline double getDecayConstant(unsigned int i) const
    {
      return _decay_constant[i];
    }
  inline double getDelayNeutronFraction(unsigned int i) const
    {
      return _delay_neutron_fraction[i];
    }
  inline double getTotalDelayNeutronFraction() const
    {
      return _total_delay_neutron_fraction;
    }
  inline double getDelaySpectrum(unsigned int i, unsigned int g) const
    {
      return _delay_chi[i*_ngroup + g];
    }
#ifdef LIBMESH_DENSE_MATRIX_H
  double k_inf_diffusion() const;
  double k_inf() const;
#endif
};
```

8. PerturbedInputXS.h

```
class PerturbedInputXS
{
public:
  PerturbedInputXS();
  void Relap5Reader(unsigned int ngroup,
                    const std::string & filename,
                    unsigned int material_id,
                    unsigned int submat_id,
                    bool dbg);
  void transposeForAdjoint();
  void convert2meter();
  void assertNotVacuum() const;
  void write(const std::string & fname, const unsigned int id=0) const;
  inline unsigned int getNG()  const;
  inline unsigned int getNParam() const;
  inline const std::vector<double>& getReferenceParams() const;
  inline unsigned int getNCoeff() const;
  inline unsigned int getNA() const;
  inline bool isFissile() const;
  inline bool hasNeutronSpeed() const;
  inline unsigned int getNumberOfDelayGroups() const;
  inline double getSigmaTotal(unsigned int g, const std::vector<double>& params) const;
  inline double getSigmaAbsorption(unsigned int g, const std::vector<double>& params) const;
  inline double getSigmaRemoval(unsigned int g, const std::vector<double>& params) const;
  inline double getDiffusionCoefficient(unsigned int g, const std::vector<double>& params) const;
  inline double getNuSigmaFission(unsigned int g, const std::vector<double>& params) const;
  inline double getSigmaFission(unsigned int g, const std::vector<double>& params) const;
  inline double getKappaSigmaFission(unsigned int g, const std::vector<double>& params) const;
  inline double getNu(unsigned int g, const std::vector<double>& params) const;
  inline double getKappa(unsigned int g, const std::vector<double>& params) const;
  inline double getFissionSpectrum(unsigned int g, const std::vector<double>& params) const;
  inline double getSigmaScattering(unsigned int gp, unsigned int g, unsigned int n,
                                   const std::vector<double>& params) const;
  inline double getSigmaScattering(unsigned int gp, unsigned int g,
                                   const std::vector<double>& params) const;
  inline double getNeutronSpeed(unsigned int g, const std::vector<double>& params) const;
  inline double getDecayConstant(unsigned int n, const std::vector<double>& params) const;
  inline double getDelayNeutronFraction(unsigned int n, const std::vector<double>& params) const;
  inline double getTotalDelayNeutronFraction(const std::vector<double>& params) const;
  inline double getDelaySpectrum(unsigned int n, unsigned int g,
                                 const std::vector<double>& params) const;
  inline double getSigmaTotal(unsigned int g) const;
  inline double getSigmaAbsorption(unsigned int g) const;
  inline double getSigmaRemoval(unsigned int g) const;
  inline double getDiffusionCoefficient(unsigned int g) const;
  inline double getNuSigmaFission(unsigned int g) const;
  inline double getSigmaFission(unsigned int g) const;
  inline double getKappaSigmaFission(unsigned int g) const;
  inline double getNu(unsigned int g) const;
  inline double getKappa(unsigned int g) const;
  inline double getFissionSpectrum(unsigned int g) const;
  inline double getSigmaScattering(unsigned int gp, unsigned int g, unsigned int n) const;
  inline double getSigmaScattering(unsigned int gp, unsigned int g) const;
  inline double getNeutronSpeed(unsigned int g) const;
  inline double getDecayConstant(unsigned int i) const;
  inline double getDelayNeutronFraction(unsigned int i) const;
  inline double getTotalDelayNeutronFraction() const;
  inline double getDelaySpectrum(unsigned int i, unsigned int g) const;
  inline double paramDiff(unsigned int i, double param) const;
};
```

9. FunctionInputXS.h

```
class Point
{
public:
  Point(double x=0., double y=0., double z=0.);
  double& x() { return _x; }
  double& y() { return _y; }
  double& z() { return _z; }
protected:
```

```
  double _x;
  double _y;
  double _z;
};
class Function
{
public:
  virtual double value(double t, const Point & p) = 0;
};
class ConstantFunction : public Function
{
public:
  ConstantFunction(double v)
      : _v(v)
    { }
  virtual double value(double, const Point &) { return _v; }
protected:
  double _v;
};
class FunctionInputXS
{
public:
  FunctionInputXS(bool ismeter=false,
                  std::vector<double> sample_t = std::vector<double>(),
                  std::vector<double> sample_p = std::vector<double>());
  ~FunctionInputXS();
  void setTotalAndScattering(unsigned int ngroup,
                             std::vector<Function *> sigt,
                             unsigned int NA,
                             std::vector<Function *> sigs);
  void setFission(unsigned int ngroup,
                  std::vector<Function *> nusigf,
                  std::vector<Function *> chi);
  void setFission1(unsigned int ngroup,
                   std::vector<Function *> sigf);
  void setFission2(unsigned int ngroup,
                   std::vector<Function *> ksigf);
  void setDiffusionRemoval(unsigned int ngroup,
                           std::vector<Function *> d,
                           std::vector<Function *> sigr,
                           std::vector<Function *> sigs);
  void setDelayNeutron(unsigned int ngroup,
                       unsigned int n_delay_groups,
                       std::vector<Function *> decay_constant,
                       std::vector<Function *> delay_neutron_fraction,
                       std::vector<Function *> delay_chi);
  void setNeutronSpeed(unsigned int ngroup,
                       std::vector<Function *> neutron_speed);
  void transposeForAdjoint();
  void assertNotVacuum() const;
  bool checkNonzeroSpectrum(unsigned int g) const;
  bool checkNonzeroScattering(unsigned int gp, unsigned int g) const;
  inline unsigned int getNG()  const;
  inline unsigned int getNA() const;
  inline bool isFissile() const;
  inline double getSigmaTotal(unsigned int g, double t, const Point &p) const;
  inline double getNuSigmaFission(unsigned int g, double t, const Point &p) const;
  inline double getSigmaFission(unsigned int g, double t, const Point &p) const;
  inline double getNu(unsigned int g, double t, const Point &p) const;
  inline double getKappaSigmaFission(unsigned int g, double t, const Point &p) const;
  inline double getKappa(unsigned int g, double t, const Point &p) const;
  inline double getFissionSpectrum(unsigned int g, double t, const Point &p) const;
  inline double getSigmaScattering(unsigned int gp, unsigned int g, unsigned int n,
                                   double t, const Point &p) const;
  inline double getSigmaScattering(unsigned int gp, unsigned int g, double t, const Point &p) const;
  inline double getNeutronSpeed(unsigned int g, double t, const Point &p) const;
  inline bool hasNeutronSpeed() const;
  inline unsigned int getNumberOfDelayGroups() const;
  inline double getDecayConstant(unsigned int i, double t, const Point &p) const;
  inline double getDelayNeutronFraction(unsigned int i, double t, const Point &p) const;
  inline double getTotalDelayNeutronFraction(double t, const Point &p) const;
  inline double getDelaySpectrum(unsigned int i, unsigned int g, double t, const Point &p) const;
  inline double getDiffusionCoefficient(unsigned int g, double t, const Point &p) const;
  inline double getSigmaRemoval(unsigned int g, double t, const Point &p) const;
  inline double getSigmaAbsorption(unsigned int g, double t, const Point &p) const;
```

```
};
```

10. Transmutation.h

```
class TransmutationLibrary
{
public:
int load1GRPXS(const std::string & fname);
int loadReactionData(const std::string & fname);
int loadFyields(const std::string & fname);
void check_isotope_chains(std::map<std::string,double> & isotope_map) const;
};
class WorkingTransmutationLibrary
{
public:
  WorkingTransmutationLibrary(const TransmutationLibrary & _TRlib,
                              std::map<std::string, double> & _initial_Number_Densities);
  void EvaluateCoefficients(std::vector<double> & consdensed_flux,
                            CSR<double> & a, std::vector< double> & b);
};
```

B Background of The Neutron Transport

B.1 The Neutron Transport Equation

The energy-dependent neutron transport equation with $\vec{r} \in \mathcal{D}$, $t \in \mathbb{R}^+$, $\vec{\Omega} \in S^2$, $E \in \mathbb{R}^+$ is given by,

$$\begin{aligned}(\frac{1}{v}\frac{\partial}{\partial t} + \vec{\Omega}\cdot\vec{\nabla} + \Sigma_t)\Psi =& S_{ext} + \frac{1}{4\pi}\int_0^\infty \chi_p(E,E')(1-\beta(E'))\nu\Sigma_f(E')\Phi(E')dE' + \\ & \int_0^\infty \int_{4\pi} \Sigma_s(E' \to E, \vec{\Omega}'\cdot\vec{\Omega})\Psi(\vec{\Omega}',E')d\Omega' dE' + \frac{1}{4\pi}\sum_{k=1}^{I} \chi_{d,k}\lambda_k C_k \end{aligned} \tag{3}$$

$$\frac{\partial C_i}{\partial t} = \int_0^\infty \beta_i(E')\nu\Sigma_f(E')\Phi(E')dE' - \lambda_i C_i, \qquad i = 1, \cdots, I \tag{4}$$

with the general boundary condition

$$\Psi(\vec{r}_b,\vec{\Omega},E,t) = \Psi^{inc}(\vec{r}_b,\vec{\Omega},E,t) + \int_0^\infty \int_{\vec{\Omega}'\cdot\vec{n}_b>0} \alpha(\vec{r}_b, E' \to E, \vec{\Omega}' \to \vec{\Omega})\Psi(\vec{r}_b,\vec{\Omega}',E',t)d\Omega' dE' \tag{5}$$

on $\vec{r}_b \in \partial\mathcal{D}$, $E \in \mathbb{R}^+$, $t \in \mathbb{R}^+$ and $\vec{\Omega}\cdot\vec{n}_b < 0$

and the initial condition

$$\Psi(\vec{r}_b,\vec{\Omega},E,t=0) = \Psi_0(\vec{r}_b,\vec{\Omega},E) \tag{6}$$

$$C_i(\vec{r}_b,t=0) = C_{i,0}(\vec{r}_b), \qquad i = 1, \cdots, I \tag{7}$$

Symbols used in the equation are standard in text. We use Σ to denote the macroscopic cross sections and reserve σ for microscopic cross sections later. Their meanings are listed below:

$\vec{r}$	position variable [cm]
$\mathcal{D}$	$\in \mathbb{R}^d$ open convex space domain, d is the spatial dimension
$\partial\mathcal{D}$	boundary of spatial domain
$\vec{n}_b$	$= \vec{n}(\vec{r}_b)$ outward unit normal vector on the boundary
$\vec{\Omega}$	angular variable
S^2	2-dimensional unit sphere
t	time variable [s]
E	energy [MeV], usually in range of [0, 20] MeV
$\mathbb{R}^+$	set of positive real number
$\Psi(\vec{r},\vec{\Omega},E,t)$	$= n(\vec{r},\vec{\Omega},E)v$ neutron density in phase space times speed also called neutron angular flux $[\frac{n}{cm^2\cdot MeV\cdot ster\cdot s}]$
$\Phi(\vec{r},E,t)$	$= \int_{4\pi}\Psi d\Omega$ neutron scalar flux $[\frac{n}{cm^2\cdot MeV\cdot s}]$
$C_i(\vec{r},t)$	fictitious delayed neutron precursor concentrations $[\frac{1}{cm^3}]$
I	number of delayed neutron precursor groups
$S_{ext}(\vec{r},\vec{\Omega},E)$	external source $[\frac{n}{cm^2\cdot MeV\cdot ster\cdot s}]$
v	$= \sqrt{\frac{2E}{m_n}}$ neutron speed$[\frac{cm}{s}]$
$\Sigma_t(\vec{r},E)$	macroscopic total cross section $[cm^{-1}]$
$\Sigma_s(\vec{r},E' \to E,\vec{\Omega}'\cdot\vec{\Omega})$	differential neutron emission or scattering cross section depending only on the cosine of scattering angles $[\frac{1}{cm\cdot MeV\cdot ster}]$
$\Sigma_f(\vec{r},E)$	fission cross section $[cm^{-1}]$
$\chi_p(\vec{r},E,E')$	prompt neutron fission spectrum$[\frac{1}{MeV}]$
$\nu(\vec{r},E)$	average number of neutrons emitted per fission
$\beta_i(\vec{r},E)$	delayed neutron fractions
$\beta(\vec{r},E)$	$= \sum_{i=1}^{I}\beta_i$ total delayed neutron fraction
$\lambda_i(\vec{r})$	decay constant of delayed neutron precursors $[\frac{1}{s}]$
$\chi_{d,i}(\vec{r},E)$	delayed neutron spectrum $[\frac{1}{MeV}]$
$\Psi^{inc}(\vec{r}_b,\vec{\Omega},E)$	incoming angular flux on the boundary $[\frac{n}{cm^2\cdot MeV\cdot ster\cdot s}]$
$\alpha(\vec{r}_b,E' \to E,\vec{\Omega}' \to \vec{\Omega})$	boundary albedo $[\frac{1}{MeV\cdot ster}]$

When $\Psi^{inc}(\vec{r}_b, \vec{\Omega}, E, t)$ is equal to zero, the boundary condition is homogeneous:

$$\begin{aligned} \Psi(\vec{r}_b, \vec{\Omega}, E, t) = \int_0^\infty \int_{\vec{\Omega}'\cdot\vec{n}_b > 0} \alpha(\vec{r}_b, E' \to E, \vec{\Omega}' \to \vec{\Omega}) \Psi(\vec{r}_b, \vec{\Omega}', E', t) d\Omega' dE' \\ \text{on } \vec{r}_b \in \partial\mathcal{D}, E \in \mathbb{R}^+ \text{ and } \vec{\Omega}\cdot\vec{n}_b < 0 \end{aligned} \tag{8}$$

In addition, when α is zero, boundary conditions are vacuum.

For reflective boundary conditions, we have $\Psi^{inc}(\vec{r}_b, \vec{\Omega}, E, t) = 0$ and

$$\alpha(\vec{r}_b, E' \to E, \vec{\Omega}' \to \vec{\Omega}) = \delta(E' - E)\delta(\vec{\Omega}' - \vec{\Omega}_r) \tag{9}$$

where

$$\vec{\Omega}_r = \vec{\Omega} - 2(\vec{\Omega}\cdot\vec{n}_b(\vec{r}_b))\vec{n}_b(\vec{r}_b). \tag{10}$$

While Σ_f is the cross section for the fission reaction only, Σ_s is the summation of cross sections of all neutron reactions with neutron re-emitted including elastic scattering, inelastic scattering and $(n, 2n)$ reaction, and etc. Σ_t is the summation of cross sections of all neutron reactions. So in view of transport equation, not all cross sections of every single neutron reaction are required.

Define following function space (not very strict here):

$$W \equiv \mathcal{D} \cup \mathbb{R}^+ \cup S^2 \cup \mathbb{R}^+ = \left\{ \Psi(\vec{r}, \vec{\Omega}, E, t) | \vec{r} \in \mathcal{D}, E \in \mathbb{R}^+, t \in \mathbb{R}^+ \text{ and } \vec{\Omega} \in S^2 \right\} \tag{11}$$

$$W^+ \equiv \partial\mathcal{D} \cup \mathbb{R}^+ \cup S^{2+} = \left\{ \Psi(\vec{r}_b, \vec{\Omega}, E) | \vec{r}_b \in \partial\mathcal{D}, E \in \mathbb{R}^+, t \in \mathbb{R}^+ \text{ and } \vec{\Omega}\cdot\vec{n}_b > 0 \right\} \tag{12}$$

$$W^- \equiv \partial\mathcal{D} \cup \mathbb{R}^+ \cup S^{2-} = \left\{ \Psi(\vec{r}_b, \vec{\Omega}, E) | \vec{r}_b \in \partial\mathcal{D}, E \in \mathbb{R}^+, t \in \mathbb{R}^+ \text{ and } \vec{\Omega}\cdot\vec{n}_b < 0 \right\} \tag{13}$$

$$V \equiv \mathcal{D} \cup \mathbb{R}^+ = \left\{ C(\vec{r}, t) | \vec{r} \in \mathcal{D} \text{ and } t \in \mathbb{R}^+ \right\} \tag{14}$$

and the following linear operators,

$$L\Psi \equiv (\vec{\Omega}\cdot\vec{\nabla} + \Sigma_t)\Psi(\vec{r}, \vec{\Omega}, E, t) \tag{15}$$

$$P\Psi \equiv \frac{1}{4\pi}\int_0^\infty \chi_p(\vec{r}, E, E')(1 - \beta(E'))\nu\Sigma_f(\vec{r}, E') \left[\int_{4\pi} \Psi(\vec{r}, \vec{\Omega}', E', t) d\Omega' \right] dE' \tag{16}$$

$$H\Psi \equiv \int_0^\infty \int_{4\pi} \Sigma_s(\vec{r}, E' \to E, \vec{\Omega}'\cdot\vec{\Omega})\Psi(\vec{r}, \vec{\Omega}', E', t) d\Omega' dE' \tag{17}$$

$$\begin{aligned} B\Psi^+ \equiv \int_0^\infty \int_{\vec{\Omega}'\cdot\vec{n}_b > 0} \beta(\vec{r}_b, E' \to E, \vec{\Omega}' \to \vec{\Omega})\Psi(\vec{r}_b, \vec{\Omega}', E', t) d\Omega' dE' \\ \text{on } \vec{r}_b \in \partial\mathcal{D} \text{ and } \vec{\Omega}\cdot\vec{n}_b < 0 \end{aligned} \tag{18}$$

$$D\mathbf{C} \equiv \frac{1}{4\pi}\sum_{j=1}^{I} \chi_{d,j}\lambda_j C_j \tag{19}$$

Ψ is a function defined on the solution space W. C_j are functions defined on the solution space V. L, P, and H are the streaming-collision, fission production and scattering operators respectively. D is the delayed neutron operator. Ψ^+ is the function defined in the space W^+. We can think it as the result of trace operation on Ψ. Similarly, Ψ^- is the result of trace operation on Ψ. B operator maps Ψ^+, all angular fluxes for outgoing directions and all energies on the boundary, to a function in the space W^-. The transport equation Eq. (3) can be written into a shorter form

$$\frac{1}{v}\frac{\partial\Psi}{\partial t} + L\Psi = S_{ext} + H\Psi + P\Psi + D\mathbf{C} \tag{20}$$

with boundary condition

$$\Psi^- = \Psi^{inc} + B\Psi^+ \tag{21}$$

Note: Ψ^{inc} is in the space W^-.

B.2 Spherical Harmonics Expansion of the Scattering Term

If we define our spherical harmonics as

$$\begin{aligned} Y_{l,m}(\vec{\Omega}) &\equiv Y^e_{l,m}(\vec{\Omega}),\ m=-l,\cdots,l;\ l=0,\cdots,n \\ Y_{n,-k}(\vec{\Omega}) &\equiv Y^o_{l,m}(\vec{\Omega}),\ m=-l,\cdots,l;\ l=1,\cdots,n \\ \Phi^g_{l,m}(\vec{r}) &\equiv \Phi^g_{n,k,e}(\vec{r}),\ m=-l,\cdots,l;\ l=0,\cdots,n \\ \Phi^g_{n,-k}(\vec{r}) &\equiv \Phi^g_{n,k,o}(\vec{r}),\ m=-l,\cdots,l;\ l=1,\cdots,n \end{aligned} \tag{22}$$

where

$$\begin{aligned} Y^e_{l,m}(\vec{\Omega}) &\equiv \sqrt{C_{l,m}}P^m_l(\mu)\cos(m\theta) \\ Y^o_{l,m}(\vec{\Omega}) &\equiv \sqrt{C_{l,m}}P^m_l(\mu)\sin(m\theta) \\ C_{l,m} &\equiv \frac{(l-m)!}{(l+m)!}(2-\delta_{m,0}) \end{aligned} \tag{23}$$

and $P^m_l(\mu)$ is the associated Legendre polynomials. They have following properties,

$$P^0_l(\mu) = P_l(\mu) \tag{24}$$

$$P^{-m}_l = (-1)^m\frac{(l-m)!}{(l+m)!}P^m_l(\mu) \tag{25}$$

$P_l(\mu)$ are the Legendre polynomials. With this definition, the addition theorem is

$$2\pi P_l(\vec{\Omega}\cdot\vec{\Omega}') = P_l(\mu_0) = \sum_{m=-l}^{l} Y_{l,m}(\vec{\Omega})Y_{l,m}(\vec{\Omega}') \tag{26}$$

We then define the angular flux moments,

$$\Phi_{l,m}(\vec{r},E,t) = \int_{4\pi}\Psi(\vec{r},\vec{\Omega},E,t)Y_{l,m}(\vec{\Omega})\,d\Omega \tag{27}$$

The angular flux can be expanded with the spherical harmonics,

$$\Psi(\vec{r},\vec{\Omega},E,t) = \sum_{n=0}^{\infty}\frac{2l+1}{4\pi}\left[\sum_{k=-n}^{n}\Phi_{l,m}(\vec{r},E,t)Y_{l,m}(\vec{\Omega})\right] \tag{28}$$

One nice thing about this definition is

$$\begin{aligned} Y_{0,0}(\vec{\Omega}) &= 1 \\ Y_{1,1}(\vec{\Omega}) &= \Omega_x \\ Y_{1,-1}(\vec{\Omega}) &= \Omega_y \\ Y_{1,0}(\vec{\Omega}) &= \Omega_z \end{aligned} \tag{29}$$

Correspondingly

$$\begin{aligned} \Phi_{0,0} &= \Phi \\ \Phi_{1,1} &= J^x \\ \Phi_{1,-1} &= J^y \\ \Phi_{1,0} &= J^z \end{aligned} \tag{30}$$

Now the scattering term

$$
\begin{aligned}
H\Psi &= \int_0^\infty \int_{4\pi} \Sigma_s(\vec{r}, E' \to E, \vec{\Omega}' \cdot \vec{\Omega}) \Psi(\vec{r}, \vec{\Omega}', E', t) \; d\Omega' dE' \\
&= \int_0^\infty \int_{4\pi} \left[\sum_{n=0}^{\infty} \frac{2n+1}{2} \Sigma_{s,n}(\vec{r}, E' \to E) P_n(\vec{\Omega}' \cdot \vec{\Omega}) \right] \cdot \left[\sum_{l=0}^{\infty} \frac{2l+1}{4\pi} \left[\sum_{m=-l}^{l} \Phi_{l,m}(\vec{r}, E', t) Y_{l,m}(\vec{\Omega}') \right] \right] \; d\Omega' dE' \\
&= \int_0^\infty \int_{4\pi} \begin{array}{c} \left[\sum_{n=0}^{\infty} \frac{2n+1}{4\pi} \Sigma_{s,n}(\vec{r}, E' \to E) \left[\sum_{k=-n}^{n} Y_{n,k}(\vec{\Omega}) Y_{n,k}(\vec{\Omega}') \right] \right] \cdot \\ \left[\sum_{l=0}^{\infty} \frac{2l+1}{4\pi} \left[\sum_{m=-l}^{l} \Phi_{l,m}(\vec{r}, E', t) Y_{l,m}(\vec{\Omega}') \right] \right] \end{array} \; d\Omega' dE' \\
&= \int_0^\infty \sum_{l=0}^{\infty} \frac{2l+1}{4\pi} \Sigma_{s,l}(\vec{r}, E' \to E) \left[\sum_{m=-l}^{l} \Phi_{l,m}(\vec{r}, E', t) Y_{l,m}(\vec{\Omega}) \right] dE'
\end{aligned}
$$

From now on, we will use this new form of scattering term.

B.3 Multigroup Transport Equations

Integrate Eq. (3) over number G of energy intervals or groups $\left[E_g, E_{g-1}\right]$, $g = 1, \cdots, G$, we obtain

$$(\frac{1}{v_g}\frac{\partial}{\partial t} + \vec{\Omega}\cdot\vec{\nabla} + \Sigma_{t,g})\Psi_g = S_{ext,g} + \frac{1}{4\pi}\sum_{g'=1}^{G}\chi_{g,g'}(1-\beta_{g'})\nu\Sigma_{f,g'}\Phi_{g'} + \sum_{g'=1}^{G}\sum_{l=0}^{\infty}\frac{2l+1}{4\pi}\sum_{m=-l}^{l}\Sigma_{s,l,m}^{g'\to g}\Phi_{g',l,m}Y_{l,m}(\vec{\Omega}) + \frac{1}{4\pi}\sum_{k=1}^{I}\chi_{d,g,k}\lambda_k C_k \tag{31}$$

where

$$\Psi_g(\vec{r},\vec{\Omega},t) \equiv \int_{E_g}^{E_{g-1}} \Psi(\vec{r},\vec{\Omega},E,t)\, dE \tag{32}$$

$$\Phi_g(\vec{r},t) \equiv \int_{E_g}^{E_{g-1}}\int_{4\pi} \Psi(\vec{r},\vec{\Omega},E,t)\, d\Omega dE \tag{33}$$

$$\Phi_{g,l,m}(\vec{r},t) \equiv \int_{E_g}^{E_{g-1}}\int_{4\pi} \Psi(\vec{r},\vec{\Omega},E,t)Y_{l,m}(\vec{\Omega})\, d\Omega dE \tag{34}$$

$$S_{ext,g}(\vec{r},\vec{\Omega},t) \equiv \int_{E_g}^{E_{g-1}} S_{ext}(\vec{r},\vec{\Omega},E,t)\, dE \tag{35}$$

$$\frac{1}{v_g(\vec{r},\vec{\Omega},t)} \equiv \frac{\int_{E_g}^{E_{g-1}} \frac{1}{v(E)}\Psi(\vec{r},\vec{\Omega},E,t)\, dE}{\int_{E_g}^{E_{g-1}} \Psi(\vec{r},\vec{\Omega},E,t)\, dE} \tag{36}$$

$$\Sigma_{t,g}(\vec{r},\vec{\Omega},t) \equiv \frac{\int_{E_g}^{E_{g-1}} \Sigma_t(\vec{r},E)\Psi(\vec{r},\vec{\Omega},E,t)\, dE}{\int_{E_g}^{E_{g-1}} \Psi(\vec{r},\vec{\Omega},E,t)\, dE} \tag{37}$$

$$\nu\Sigma_{f,g}(\vec{r},t) \equiv \frac{\int_{E_g}^{E_{g-1}} \nu\Sigma_f(\vec{r},E)\Phi(\vec{r},E,t)\, dE}{\int_{E_g}^{E_{g-1}} \Phi(\vec{r},E,t)\, dE} \tag{38}$$

$$\beta_{i,g}(\vec{r},t) \equiv \frac{\int_{E_g}^{E_{g-1}} \beta_i(\vec{r},E)\nu\Sigma_f(\vec{r},E)\Phi(\vec{r},E,t)\, dE}{\int_{E_g}^{E_{g-1}} \nu\Sigma_f(\vec{r},E)\Phi(\vec{r},E,t)\, dE} \tag{39}$$

$$\beta_g(\vec{r},t)) = \sum_{i=0}^{I}\beta_{i,g}(\vec{r},t) \tag{40}$$

$$\chi_{g,g'}(\vec{r},t) \equiv \frac{\int_{E_g}^{E_{g-1}}\int_{E_{g'}}^{E_{g'-1}} \chi(\vec{r},E,E')(1-\beta(\vec{r},E'))\nu\Sigma_f(E')\Phi(\vec{r},E',t)\, dE'\, dE}{\int_{E_{g'}}^{E_{g'-1}}(1-\beta(\vec{r},E'))\nu\Sigma_f(\vec{r},E')\Phi(\vec{r},E',t)\, dE'} \tag{41}$$

$$\Sigma_{s,n,k}^{g'\to g}(\vec{r},t) \equiv \frac{\int_{E_{g'}}^{E_{g'-1}}\left[\int_{E_g}^{E_{g-1}} \Sigma_{s,n}(\vec{r},E'\to E)\, dE\right]\Phi_{l,m}(\vec{r},E',t)\, dE'}{\int_{E_{g'}}^{E_{g'-1}} \Phi_{l,m}(\vec{r},E',t)\, dE'} \tag{42}$$

and

$$\chi_{d,g,k}(\vec{r}) = \int_{E_g}^{E_{g-1}} \chi_{d,k}(\vec{r},E)\, dE \tag{43}$$

The delayed neutron precursor equation becomes

$$\frac{\partial C_i}{\partial t} = \sum_{g'=1}^{G}\beta_{i,g'}\nu\Sigma_{f,g'}\Phi_{g'} - \lambda_i C_i, \qquad i = 1,\cdots,I \tag{44}$$

By definition

$$\sum_{g=1}^{G} \chi_{g,g'}(\vec{r},t) \equiv 1. \tag{45}$$

Note that the group-averaged total cross sections is angular dependent. We can apply similar spherical harmonics expansion to the total collision term,

$$\begin{aligned}
&\int_{E_g}^{E_{g-1}} \Sigma_t(\vec{r},E)\Psi(\vec{r},\vec{\Omega},E,t)\,dE \\
&= \sum_{l=0}^{\infty} \frac{2l+1}{4\pi} \sum_{m=-l}^{l} \int_{E_g}^{E_{g-1}} \Sigma_t(\vec{r},E)\Phi_{l,m}(\vec{r},E,t)Y_{l,m}(\vec{\Omega})\,dE \\
&= \sum_{l=0}^{\infty} \frac{2l+1}{4\pi} \sum_{m=-l}^{l} \Sigma_{t,g,l,m}(\vec{r},t)\Phi_{g,l,m}(\vec{r},t)Y_{l,m}(\vec{\Omega})
\end{aligned}$$

where

$$\Sigma_{t,g,l,m}(\vec{r},t) \equiv \frac{\int_{E_g}^{E_{g-1}} \Sigma_t(\vec{r},E)\Phi_{l,m}(\vec{r},E,t)\,dE}{\int_{E_g}^{E_{g-1}} \Phi_{l,m}(\vec{r},E,t)\,dE} \tag{46}$$

Note that it is possible $\Phi_{g,l,m}(\vec{r},t)$ is equal to zero. In such a case, $\Sigma_{t,g,l,m}(\vec{r},t)$ is not defined and not needed. We can however define

$$\Sigma_{t,g}(\vec{r},t) \equiv \Sigma_{t,g,0,0}(\vec{r},t) \tag{47}$$

and modify the in-group scattering cross section to keep the equation valid

$$\Sigma_{s,l,m}^{g'\to g}(\vec{r},t) \leftarrow \Sigma_{s,l,m}^{g'\to g}(\vec{r},t) + (\Sigma_{t,g}(\vec{r},t) - \Sigma_{t,g,l,m}(\vec{r},t))\delta_{l0}\delta_{m0}\delta_{gg'} \tag{48}$$

The above equations state the exact projection operation in order that the multigroup equation preserves the solution of the energy-dependent transport equation. Quantities including $v_g, \Sigma_{t,g}, \nu\Sigma_{f,g}, \beta_{i,g}, \chi_{g,g'}, \Sigma_{s,n,k}^{g'\to g}, \chi_{d,g,j}$ are called group constants.

The above equations also mean that we can obtain the multigroup cross sections exactly only after we know the continuous transport solution. However, the above equations allow us introduce the multigroup approximation:

$$\Psi(\vec{r},\vec{\Omega},E,t) = \hat{\Psi}(\vec{r},\vec{\Omega},t)f_{g,r}(E) \tag{49}$$

so

$$\Phi(\vec{r},E,t) = \hat{\Phi}(\vec{r},t)f_{g,r}(E) \tag{50}$$

r is the region ID under consideration. We may have many regions in the solution domain to reduce the approximation error induced by the multigroup approximation. $f_{g,r}(E)$ is called as the neutron spectrum of a region r and an energy group g. These spectrum are obtained by doing fine-group calculations with proper resonance treatment with some small reference configurations. We assume these reference configurations can

produce the spectrum sufficiently accurate for the core level simulations. With this approximation,

$$\Sigma_{t,g}(\vec{r}) \equiv \frac{\int_{E_g}^{E_{g-1}} \Sigma_t(\vec{r},E) f_{g,r}(E)\, dE}{\int_{E_g}^{E_{g-1}} f_{g,r}(E)\, dE} \tag{51}$$

$$\nu\Sigma_{f,g}(\vec{r}) \equiv \frac{\int_{E_g}^{E_{g-1}} \nu\Sigma_f(\vec{r},E) f_{g,r}(E)\, dE}{\int_{E_g}^{E_{g-1}} f_{g,r}(E)\, dE} \tag{52}$$

$$\Sigma_{s,l}^{g'\to g}(\vec{r}) \equiv \frac{\int_{E_{g'}}^{E_{g'-1}} \left[\int_{E_g}^{E_{g-1}} \Sigma_{s,l}(\vec{r},E'\to E)\, dE\right] f_{g',r}(E')\, dE'}{\int_{E_{g'}}^{E_{g'-1}} f_{g',r}(E')\, dE'} \tag{53}$$

$$\frac{1}{v_g(\vec{r})} \equiv \frac{\int_{E_g}^{E_{g-1}} \frac{1}{v(E)} f_{g,r}(E)\, dE}{\int_{E_g}^{E_{g-1}} f_{g,r}(E)\, dE}. \tag{54}$$

We expect when the regions covering the domain are getting smaller and energy groups are becoming thinner, the error caused by this approximation tends to be zero (questionable). The accuracy of the multigroup approximation depends on how the full range of energy is cut and number of regions are considered and also depends on how well the spectrum can be obtained.

Comparing with the exact projection,

- the scattering cross section is now only l-dependent;
- the averaged neutron velocity is angular independent;
- all group constants have no time dependency.

If we neglect the energy dependency of the prompt fission spectrum χ_p, then we simply have

$$\chi_g(\vec{r}) = \int_{E_g}^{E_{g-1}} \chi(\vec{r},E)\, dE. \tag{55}$$

Similarly if we neglect the energy dependency of the delayed neutron fractions, there will be no group averaging of these fractions.

The balance equation can be obtained by integrating the multigroup transport equation in the 2D unit sphere:

$$\frac{1}{v_{g,0}}\frac{\partial \Phi_g}{\partial t} + \vec{\nabla}\cdot\mathbf{J}_g + \Sigma_{t,g}\Phi_g = Q_{ext,g} + \sum_{g'=1}^{G}\chi_{g,g'}(1-\beta_{g'})\nu\Sigma_{f,g'}\Phi_{g'} + \sum_{g'=1}^{G}\Sigma_s^{g'\to g}\Phi_{g'} + \sum_{k=1}^{I}\chi_{d,g,k}\lambda_k C_k \tag{56}$$

Similarly the current equation can be obtained by integrating the multigroup transport equation multiplied with the firs-order spherical harmonics in the 2D unit sphere:

$$\frac{1}{v_{g,1}}\frac{\partial \mathbf{J}}{\partial t} + \frac{1}{3}\vec{\nabla}\Phi_g + \Sigma_{t,g}\mathbf{J}_g = \mathbf{Q}_{1,ext,g} + \sum_{g'=1}^{G}\Sigma_{s,1}^{g'\to g}\mathbf{J}_{g'} \tag{57}$$

By assume zero time-derivative term and zero external anisotropic source, the above equation reduced to

$$\frac{1}{3}\vec{\nabla}\Phi_g + \Sigma_{t,g}\mathbf{J}_g = \sum_{g'=1}^{G}\Sigma_{s,1}^{g'\to g}\mathbf{J}_{g'}. \tag{58}$$

With the transport approximation

$$\Sigma_{s,1}^{g'\to g}\mathbf{J}_{g'} = \Sigma_{s,1}^{g\to g'}\mathbf{J}_g, \tag{59}$$

we are able to get the following Fick's law

$$\mathbf{J}_g = -\frac{1}{3(\Sigma_{t,g} - \sum_{g'=1}^{G} \Sigma_{s,1}^{g\to g'})}\vec{\nabla}\Phi_g. \tag{60}$$

We usually define the transport cross section as

$$\Sigma_{tr,g} \equiv \Sigma_{t,g} - \sum_{g'=1}^{G} \Sigma_{s,1}^{g\to g'}. \tag{61}$$

With the definition, the diffusion coefficient is

$$D_g \equiv \frac{1}{3\Sigma_{tr,g}}. \tag{62}$$

If we substitute Eq. (60) into the balance equation, we obtain the multigroup diffusion equation.

Higher order ($l > 1$) transport cross section is extended as

$$\Sigma_{tr,l} \equiv \Sigma_{t,g,l} - \sum_{g'=1}^{G} \Sigma_{s,l}^{g\to g'}, \qquad l = 2, \cdots, \infty \tag{63}$$

B.4 Mixing

For all reactions of mixture, the point-wise macroscopic cross sections satisfy

$$\Sigma_x(\vec{r}, E, t) = \sum_{k=1}^{M} N_k(\vec{r}, t)\sigma_{x,k}(E) \tag{64}$$

where k is the isotope index; M is the number of isotopes in the mixture; $N_k, k = 1, \cdots, M$ are the isotope atom densities; σ represents the microscopic cross sections. The same for all secondary particle emissions

$$\Sigma_x(\vec{r}, E' \to E, \vec{\Omega} \cdot \vec{\Omega}', t) = \sum_{k=1}^{M} N_k(\vec{r}, t)\sigma_{x,k}(E')p_k(E' \to E, \vec{\Omega} \cdot \vec{\Omega}') \tag{65}$$

where p_k is the differential distribution of the secondary particles. Note that the differential distribution is isotope dependent.

With group integration, the group constants $\Sigma_{t,g}, \nu\Sigma_{f,g}, \Sigma_{s,n,k}^{g'\to g}$ can be easily obtained with mixing because only flux shows up in the denominator of their definition.

$$\Sigma_{t,g}(\vec{r}, t) = \sum_{k=1}^{M} N_k(\vec{r}, t)\sigma_{t,g,k} \tag{66}$$

$$\nu\Sigma_{f,g}(\vec{r}, t) = \sum_{k=1}^{M} N_k(\vec{r}, t)\nu\vartheta_{f,g,k} \tag{67}$$

$$\Sigma_{s,l,m}^{g'\to g}(\vec{r}, t) = \sum_{k=1}^{M} N_k(\vec{r}, t)\sigma_{s,l,m,k}^{g'\to g} \tag{68}$$

Now σ are the group-averaged microscopic cross sections, which can be obtained separately from the mixing.

However, $\beta_{i,g}$ and $\chi_{g,g'}$ requires the mixing also in the denominator. Although the best way is to define several new quantities like

$$\beta\nu\Sigma_{f,i,g}(\vec{r}, t) \equiv \frac{\int_{E_g}^{E_{g-1}} \beta_i(\vec{r}, E)\nu\Sigma_f(\vec{r}, E)\Phi(\vec{r}, E, t)\, dE}{\int_{E_g}^{E_{g-1}} \Phi(\vec{r}, E, t)\, dE} \tag{69}$$

$$\chi_{g,g'}(\nu\Sigma_{f,g} - \beta\nu\Sigma_{f,g})(\vec{r}, t) \equiv \frac{\int_{E_g}^{E_{g-1}} \int_{E_{g'}}^{E_{g'-1}} \chi(\vec{r}, E, E')(1 - \beta(\vec{r}, E'))\nu\Sigma_f(E')\Phi(\vec{r}, E', t)\, dE'\, dE}{\int_{E_{g'}}^{E_{g'-1}} \Phi(\vec{r}, E', t)\, dE'} \tag{70}$$

and do the mixing with these new-defined quantities and then calculate

$$\beta_{i,g}(\vec{r}, t) = \frac{\beta\nu\Sigma_{f,i,g}(\vec{r}, t)}{\nu\Sigma_{f,g}(\vec{r}, t)} \tag{71}$$

$$\chi_{g,g'}(\vec{r}, t) = \frac{\chi_{g,g'}(\nu\Sigma_{f,g} - \beta\nu\Sigma_{f,g})(\vec{r}, t)}{\nu\Sigma_{f,g} - \beta\nu\Sigma_{f,g}}, \tag{72}$$

we usually apply the following approximation

$$\beta_{i,g} = \frac{\sum_{k,\beta_{i,g,k}\neq 0} N_k\beta_{i,g,k}}{\sum_{k,\beta_{i,g,k}\neq 0} N_k} \tag{73}$$

$$\chi_{g,g'} = \frac{\sum_{k,\chi_{g,g',k}\neq 0} N_k\chi_{g,g',k}}{\sum_{k,\chi_{g,g',k}\neq 0} N_k} \tag{74}$$

Other quantities are irrelevant to the mixing like the averaged neutron speed and the delayed neutron spectrum. For the neutron speed, we often assume it is angle independent and equal to

$$v_g(\vec{r}, t) = \frac{\int_{E_g}^{E_{g-1}} \Phi(\vec{r}, E, t)\ dE}{\int_{E_g}^{E_{g-1}} \frac{1}{v(E)} \Phi(\vec{r}, E, t)\ dE}. \tag{75}$$

Sometimes people also assume the delayed neutron fraction and prompt neutron spectrum is isotope independent, then these quantities become irrelevant to the mixing as well.

So in summary, we have three kinds of properties:

1. normal mixing properties;
2. atomic density weighted properties;
3. non-mixing properties.

Different kinds of properties require different processing of mixing for multigroup calculations.

B.5 Depletion

The governing equation for depletion is

$$\frac{dN_j}{dt} = \sum_{k=1}^{M} (\sum_{g=1}^{G} \gamma_j \sigma_{f,g,k} \Phi_g + \sum_{g=1}^{G} \sigma_{g,k\to j} \Phi_g + r_{k\to j}\lambda_k) N_k - (\sum_{g=1}^{G} \sigma_{d,g,j}\Phi_g) N_j - \lambda_j N_j, \qquad j = 1, \cdots, M \tag{76}$$

Symbols used in the equation are standard in text, their meanings are listed below:

$N_j(\vec{r}, t)$	atom density of isotope $j[\frac{1}{barn \cdot cm}]$
M	number of isotopes in consideration
γ_j	fission production yield of isotope j
$\sigma_{f,g,k}$	fission cross section of isotope $k[barn]$
$\Phi_g(\vec{r}, t)$	neutron flux $[\frac{1}{barn \cdot s}]$
$\sigma_{g,k\to j}$	transmutation into j from isotope k $[barn]$
$r_{k\to j}$	branch ratio of decay of k creating j
$\sigma_{d,g,j}$	destruction of j due to neutron reactions $[barn]$
λ_j	decay constant of j $[\frac{1}{s}]$

Number of energy groups for depletion may not be the same as the one for transport calculation. CASMO-4 use 70 number of energy groups while HELIOS use 45 groups. (more reference) Usually about 40 actinides and over 100 fission products need to be tracked to obtain a good accuracy. Because there is no spatial derivative terms in the equation, which means the equation can be solved independently on spatial points, we will drop the spatial dependency of N_j from now on.

The equation can be written in vector form:

$$\frac{d\mathbf{N}(t)}{dt} = \mathbf{A}(t)\mathbf{N}, \tag{77}$$

where the matrix $\mathbf{A}$ is called the transition matrix. Its dependency on time comes from the time dependency of scalar fluxes and the microscopic cross sections due to spectrum shift. We usually use the *predictor-corrector* technique to match from the beginning to the end of a time step $[t_{n-1}, t_n]$:

1. Prediction:
$$\mathbf{N}^p(t_n) = \mathbf{N}(t_{n-1}) e^{\mathbf{A}(t_{n-1})\Delta t}; \tag{78}$$
2. Update the microscopic cross sections and fluxes with a transport solve, and obtain the predicted transition matrix $\mathbf{A}^p(t_n)$;
3. Correction:
$$\mathbf{N}^c(t_n) = \mathbf{N}(t_{n-1}) e^{\mathbf{A}^p(t_n)\Delta t} \tag{79}$$
4. Averaging the two atom densities with
$$\mathbf{N}(t_n) = \frac{\mathbf{N}^p(t_n) + \mathbf{N}^c(t_n)}{2}. \tag{80}$$
5. Update the microscopic cross sections and fluxes with a transport solve, and obtain the predicted transition matrix $\mathbf{A}(t_n)$;

We notice that every depletion time step, two transport solves are required. Predictor-corrector technique can better account for the nonlinear behavior of transition matrices then the middle-timestep approach.

Decay heat build-up needs to be taken into account. The averaged decay heat percentage of total fission energy per fission counted from the fission event follows the following empirical rule:

$$p_d(t) = 1.36(t + 1)^{-1.2} \tag{81}$$

So the decay heat built-up can be evaluated by

$$P_d(t) = \int_0^t P(t')1.36(t - t' + 1)^{-1.2} dt'. \tag{82}$$

At every time step, the prompt power release must be calculated by

$$P_p(t) = P(t) - P_d(t) \tag{83}$$

and can be used to normalize the fluxes by assuring

$$\int_{\mathcal{D}} \sum_{g=1}^{G} \kappa \Sigma_{f,g} \Phi_g \, d\vec{r} = P(t) - P_d(t). \tag{84}$$

Note κ is the prompt energy release.

Questions could be raised: where the cross section is obtained? Xenon migration?

Processing codes including NJOY include functions such as resonance reconstruction, Doppler broadening, multigroup averaging, and/or rearrangement into specified interface formats. Their multigroup averaging is not sufficient to account for all the self-shielding effects!

C Isotope Identifier (MAT)

Contents of this section are obtained from the following two documents available on Internet:

- ENDF-6 Formats Manual (http://www-nds.iaea.org/ndspub/documents/endf/endf102/endf102.pdf)
- An Introduction to the ENDF Formats (http://t2.lanl.gov/endf/title.html)

Readers are referred to them for further information.

Each material in an ENDF library is assigned a unique identification number, designated by the symbol MAT, which ranges from 1 to 9999.

One hundred MAT numbers (Z01-Z99) have been allocated to each element Z, through Z = 98. Natural elements have MAT numbers Z00. The MAT numbers for isotopes of an element has format Zxx, where xx are assigned on the basis of increasing mass in steps of three, allowing for the ground state and two metastable states. (This procedure leads to difficulty for the nuclides of xenon, cesium, osmium, platinum, etc., where more than 100 MAT numbers could be needed and some decay data where more than two isomeric states might be present.)

A complete index to ENDF/B-VI neutron data can be found at "An Introduction to the ENDF Formats" http://t2.lanl.gov/cgi-bin/nuclides/endind.

For mixtures, compounds, alloys, and molecules, MAT numbers between 0001 and 0099 are assigned on a special basis. The presently recognized assignments are:

Table 5 Recognized compound MAT numbers.

Compound	MAT Number
Water	1
Para Hydrogen	2
Ortho Hydrogen	3
H in ZrH	7
Heavy Water	11
Para Deuterium	12
Ortho Deuterium	13
Be	26
BeO	27
Be2C	28
Be in BeO	29
Graphite	31
Liquid Methane	33
Solid methane	34
Polyethylene	37
Benzene	40
O in BeO	46
Zr in ZrH	58
UO2	75
UC	76

The MAT numbers can be directly adopted for identifying isotopes.

D INSTANT XML Format for Macroscopic Cross Sections

INSTANT XML format provides a convenient way of describing a collection of macroscopic cross sections.

D.1 *Macros*

Attributes of Macros:

- *Name*
  Description: the name of this set of macroscopic cross sections
  Data type: string
  Default value: an empty string
- *NG*
  Description: the number of energy groups
  Data type: integer
  Default value: N/A
- *I*
  Description: the number of delayed neutron groups
  Data type: integer
  Default value: 0

Arbitrary number of materials can be contained in Macros. Each has the following format.

D.1.1 *material*

Attributes of material:

- *ID*
  Description: the unique material ID (> 0)
  Data type: integer
  Default value: N/A
- *NA*
  Description: the order of scattering anisotropy (≥ 0) (0 means isotropic scattering)
  Data type: integer
  Default value: 0
- *fissile*
  Description: to indicate if the material is fissile
  Data type: logical
  Default value: false

Elements of material:

- *name* – a short description of the material.
- *TotalXS* – total cross sections of all groups $\sigma_{t,g}$. The unit is $1/cm$.
- *NuFissionXS* – fission cross sections times the averaged neutrons emitted per fission $\nu\sigma_{f,g}$. The unit is $1/cm$. Only fissile materials can and must have this cross section.
- *FissionXS* – fission cross sections $\sigma_{f,g}$. The unit is $1/cm$. Only fissile materials can have this cross section.

- *KappaFissionXS* – energy deposited per fission times the fission cross sections $\kappa\sigma_{f,g}$. The unit is J/cm. Only fissile materials can have this cross section.

- *ChiXS* – fission spectrum χ_g. Only fissile materials can and must have this cross section. The summation of the fission neutron spectrum for all energy groups must be equal to one. When the number of delayed neutron groups I is non-zero, fission spectrum given should be the prompt fission spectrum, otherwise it should be the averaged neutron spectrum.

- *Profile* – profile of the scattering matrix $\sigma_{s,n}^{g'\to g}, g' = 1, \cdots, NG; g = 1, \cdots, NG; n = 0, \cdots, NA$. *Profile* is designed to save the input effort and the size of the file containing the scattering cross sections. There are $NG \times (NA+1)$ pairs of integers. All these pairs are ordered by energy index g first then anisotropy group n, i.e. $((g = 1, G), n = 0, NA)$. Each pair has the first departure scattering group and the last departure scattering group for the anisotropy n and the group g. *Profile* is used to reduce the amount of inputs for the scattering matrix. Users do not have to give *Profile*. If *Profile* is absent, all entries of the scattering matrix must be inputted including all zero entries. This also means the default value of all pairs of *Profile* is $(1, NG)$. For example, the following 7-by-7 scattering matrix where non-zeros are marked with $\times$,

$$\begin{array}{c} \to g' \\ \downarrow g \left[\begin{array}{ccccccc} \times & & & & & & \\ \times & \times & & & & & \\ \times & \times & \times & & & & \\ & & \times & \times & \times & & \\ & & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times & \times \end{array} \right]_{7\times 7} \end{array}$$

  has *Profile* $[1 \quad 1 \quad\quad 1 \quad 2 \quad\quad 1 \quad 3 \quad\quad 3 \quad 5 \quad\quad 3 \quad 7 \quad\quad 3 \quad 7 \quad\quad 3 \quad 7]$.

- *ScatteringXS* – the $NA+1$ scattering matrix $\sigma_{s,n}^{g'\to g}$. The unit is $1/cm$. All scattering matrices are inputted sequentially. For each scattering matrix, entries specified by *Profile* are inputted row by row. The column index of the scattering matrix is g'; the row index of the scattering matrix is g. If there is no up-scatterings, the scattering matrix is strictly low-triangular. Note that INSTANT treats all groups without up-scatterings as the "fast" group and all groups with up-scatterings as the "thermal" group. "fast" here does not necessarily mean the energy of neutrons in the group is high. Thermal iteration may be required for solving problems with up-scattering materials. Every material must have the scattering cross sections.

- *DiffusionCoefficient* – diffusion coefficients of all groups D_g. The unit is cm.

- *RemovalXS* – removal cross sections of all groups $\sigma_{r,g}$. The unit is $1/cm$.

- *NeutronSpeed* – averaged neutron speeds of all groups v_g. The unit is cm/s. It must present for transient calculations.

- *DNFraction* – delayed neutron fraction of all delayed groups β_i.

- *DNPlambda* – decay constant of delayed neutron precursors of all delayed groups λ_i. The unit is $1/s$.

- *DNSpectrum* – decay neutron spectrum of all delayed groups $\chi_{i,g}$.

- *AbsorptionXS* – absorption cross sections of all groups $\sigma_{a,g}$. The unit is $1/cm$.

Note that all materials must have the total cross sections for the transport calculations and all materials must have the diffusion coefficients for the diffusion calculations. In-group scattering cross sections are not necessary for diffusion calculations when the removal cross sections are provided.

E INSTANT XML Format for Perturbed Macroscopic Cross Sections

INSTANT XML format provides a convenient way of describing a collection of perturbed macroscopic cross sections.

E.1 *Macros*

Attributes of Macros:

- *Name*
  Description: the name of this set of macroscopic cross sections
  Data type: string
  Default value: an empty string
- *NG*
  Description: the number of energy groups
  Data type: integer
  Default value: N/A
- *I*
  Description: the number of delayed neutron groups
  Data type: integer
  Default value: 0

Arbitrary number of materials can be contained in Macros. Each has the following format.

E.1.1 *material*

Attributes of material1:

- *ID*
  Description: the unique material ID (> 0)
  Data type: integer
  Default value: N/A
- *name*
  Description: the name of the material
  Data type: string
  Default value: N/A
- *NA*
  Description: the order of scattering anisotropy (≥ 0) (0 means isotropic scattering)
  Data type: integer
  Default value: 0
- *fissile*
  Description: to indicate if the material is fissile
  Data type: logical
  Default value: false

Elements of material are Pattern, Reference and Perturbations.

E.1.2 *Pattern*

This element describes how the cross sections are perturbed.

- *NParameters* – number of perturbation parameters.
- *ReferParameters* – reference values of perturbation parameters. Number of values must be equal to *NParameters*.
- *ParameterOrders* – perturbation orders of all parameters. orders are greater or equal to one. Number of values must be equal to *NParameters*.
- *ParameterTypes* – perturbation type of all parameters. Valid values include LINEAR and SQRT. Number of values must be equal to *NParameters*.
- *TotalXS* – perturbation pattern of total cross sections. The valid value is either *perturbed* or *fixed*. By default, the value is *fixed*.
- *NuFissionXS* – perturbation pattern of ν fission cross sections. By default, the value is *fixed*.
- *FissionXS* – perturbation pattern of fission cross sections. By default, the value is *fixed*.
- *KappaFissionXS* – perturbation pattern of energy deposited per fission times the fission cross sections. By default, the value is *fixed*.
- *ChiXS* – perturbation pattern of fission spectrum. By default, the value is *fixed*.
- *ScatteringXS* – perturbation pattern of scattering cross sections. By default, the value is *fixed*.
- *DiffusionCoefficient* – perturbation pattern of diffusion coefficients. By default, the value is *fixed*.
- *RemovalXS* – perturbation pattern of removal cross sections. By default, the value is *fixed*.
- *NeutronSpeed* – perturbation pattern of averaged neutron speeds. By default, the value is *fixed*.
- *DNFraction* – perturbation pattern of delayed neutron fractions. By default, the value is *fixed*.
- *DNPlambda* – perturbation pattern of decay constants. By default, the value is *fixed*.
- *DNSpectrum* – perturbation pattern of decay neutron spectrum. By default, the value is *fixed*.
- *AbsorptionXS* – perturbation pattern of absorption cross sections. By default, the value is *fixed*.

E.1.3 *Reference*

This element contains all the cross sections at the reference state.

- *TotalXS* – total cross sections of all groups $\sigma_{t,g}$. The unit is $1/cm$.
- *NuFissionXS* – fission cross sections times the averaged neutrons emitted per fission $\nu\sigma_{f,g}$. The unit is $1/cm$. Only fissile materials can and must have this cross section.
- *FissionXS* – fission cross sections $\sigma_{f,g}$. The unit is $1/cm$. Only fissile materials can have this cross section.
- *KappaFissionXS* – energy deposited per fission times the fission cross sections $\kappa\sigma_{f,g}$. The unit is J/cm. Only fissile materials can have this cross section.
- *ChiXS* – fission spectrum χ_g. Only fissile materials can and must have this cross section. The summation of the fission neutron spectrum for all energy groups must be equal to one. When the number of delayed neutron groups I is non-zero, fission spectrum given should be the prompt fission spectrum, otherwise it should be the averaged neutron spectrum.

- *Profile* – profile of the scattering matrix $\sigma_{s,n}^{g' \to g}, g' = 1, \cdots, NG; g = 1, \cdots, NG; n = 0, \cdots, NA$. *Profile* is designed to save the input effort and the size of the file containing the scattering cross sections. There are $NG \times (NA + 1)$ pairs of integers. All these pairs are ordered by energy index g first then anisotropy group n, i.e. $((g = 1, G), n = 0, NA)$. Each pair has the first departure scattering group and the last departure scattering group for the anisotropy n and the group g. *Profile* is used to reduce the amount of inputs for the scattering matrix. Users do not have to give *Profile*. If *Profile* is absent, all entries of the scattering matrix must be inputted including all zero entries. This also means the default value of all pairs of *Profile* is $(1, NG)$. For example, the following 7-by-7 scattering matrix where non-zeros are marked with $\times$,

$$
\begin{array}{c}
\rightarrow g' \\
\downarrow_g \begin{bmatrix}
\times & & & & & & \\
\times & \times & & & & & \\
\times & \times & \times & & & & \\
 & & \times & \times & \times & & \\
 & & \times & \times & \times & \times & \times \\
 & & \times & \times & \times & \times & \times \\
 & & \times & \times & \times & \times & \times
\end{bmatrix}_{7\times 7}
\end{array}
$$

  has *Profile* $[1 \quad 1 \qquad 1 \quad 2 \qquad 1 \quad 3 \qquad 3 \quad 5 \qquad 3 \quad 7 \qquad 3 \quad 7 \qquad 3 \quad 7]$.

- *ScatteringXS* – the $NA + 1$ scattering matrix $\sigma_{s,n}^{g' \to g}$. The unit is $1/cm$. All scattering matrices are inputted sequentially. For each scattering matrix, entries specified by *Profile* are inputted row by row. The column index of the scattering matrix is g'; the row index of the scattering matrix is g. If there is no up-scatterings, the scattering matrix is strictly low-triangular. Note that INSTANT treats all groups without up-scatterings as the "fast" group and all groups with up-scatterings as the "thermal" group. "fast" here does not necessarily mean the energy of neutrons in the group is high. Thermal iteration may be required for solving problems with up-scattering materials. Every material must have the scattering cross sections. This element can have an logical attribute *has2l* with default value false to indicate if the scattering cross sections have the $2l + 1$ factor.
- *DiffusionCoefficient* – diffusion coefficients of all groups D_g. The unit is cm.
- *RemovalXS* – removal cross sections of all groups $\sigma_{r,g}$. The unit is $1/cm$.
- *NeutronSpeed* – averaged neutron speeds of all groups v_g. The unit is cm/s. It must present for transient calculations.
- *DNFraction* – delayed neutron fraction of all delayed groups β_i.
- *DNPlambda* – decay constant of delayed neutron precursors of all delayed groups λ_i. The unit is $1/s$.
- *DNSpectrum* – decay neutron spectrum of all delayed groups $\chi_{i,g}$.
- *AbsorptionXS* – absorption cross sections of all groups $\sigma_{a,g}$. The unit is $1/cm$.

Note that all materials must have the total cross sections for the transport calculations and all materials must have the diffusion coefficients for the diffusion calculations. In-group scattering cross sections are not necessary for diffusion calculations when the removal cross sections are provided.

E.1.4 *Perturbation*

There could be multiple copies of this element. Each of them corresponds to one term of the perturbation in Eq. (1) or Eq. (2), which are specified by two attributes:

- *perturbing*

  Description: the index of the perturbed parameter

Data type: integer

Default value: N/A

Note: the index starts from one.

- *order*

  Description: the order of this perturbation

  Data type: integer

  Default value: N/A

  Note: the value must be smaller or equal to the order of the parameter in Pattern. It also must be greater than zero.

This element contains all the perturbation coefficients. It has the exactly same sub-elements of Reference, when the corresponding cross section is perturbed. The values are the perturbation coefficients though.

References

[1] D. Kent Parsons. *ANISN/PC Manual*. EG&G Idaho, Inc., December 1988.

[2] R. Douglas O'Dell. Standard interface files and procedures for reactor physics codes, version IV. Technical Report LA-6941-MS, Los Alamos Scientific Laboratory, 1977.

[3] M. E. Dunn and N. M. Greene. AMPX-2000: Cross-section processing system for generating nuclear data for criticality safety applications. *Trans. Am. Nucl. Soc.*, 86:118–119, 2002.

[4] http://www.w3.org/xml/.

[5] Yaqi Wang. *INSTANT User's Manual*. INL, 2012.

[6] Derek Gaston, Chris Newman, Glen Hansen, and Damien Lebrun-Grandie. Moose: A parallel computational framework for coupled systems of nonlinear equations. *Nuclear Engineering and Design*, 239(10):1768–1778, 2009.

[7] http://www.unidata.ucar.edu/downloads/netcdf/index.jsp.

[8] http://rapidxml.sourceforge.net/index.htm.

[9] The Members of the Cross Section Evaluation Working Group. ENDF-6 formats manual. Research Report 44945-05-Rev, National Nuclear Data Center, Brookhaven National Laboratory, Upton, N.Y., 2005.