LA-UR-22-31848

Approved for public release; distribution is unlimited.

Title: The Intrinsic Source Constructor Package: Installation and Use

Author(s): Solomon, Clell Jeffrey Jr.
Bates, Cameron Russell
Kulesza, Joel A.
Marcath, Matthew James

Intended for: Report

Issued: 2022-11-08

The Intrinsic Source Constructor Package: Installation and Use

Clell J. (CJ) Solomon Jr., Cameron R. Bates, Joel A. Kulesza, and Matthew J. Marcath

Los Alamos National Laboratory, X-Computational Physics Division

October 28, 2022

Contents

1 2.1.0 Release Notes

The main changes in the ISC 2.1.0 release are:

- CMake refactored for better Windows support (including multi-build)
- Exported target names in CMake are now in the `isc` namespace
- Headers moved into `isc` sub-directory
- Python deployment integrated into CMake build process
- ENDF-VIII.0 data libraries added
- Proton decay fixed (`A` was not decremented for daughter)
- Cf-252 neutron intensity reduced by a factor of 31.4 (branching ratio was calculated incorrectly previously)
- Additional biasing features in MISC (noted in the updated MISC users guide which can be found in `utils/misc/userguide` in the ISC source tree)

2 Installation

2.1 Overview and Requirements

The Intrinsic Source Constructor (ISC) library is a C++ software library bound to Python 3 via the Simplified Wrapper and Interface Generator (SWIG version 3.0.7). The minimum requirements to build ISC as a C++ library are the following:

- a C++ compiler supporting C++11 features (most modern compilers support this standard)
- the CMake tool set version 3.14 or greater

Currently, the following compilers are supported:

- GCC 5.3.0 and above on Linux and macOS
- Apple Clang 7.3.0 and above on macOS
- Microsoft Visual Studio 2022 on Windows

Additionally, one must have Python 3 installed to build the Python bindings. CMake is not required should one desire to build only the Python components.

2.2 Building the ISC C++ Library and Utilities

To build ISC, start by creating a directory to build it. Within the directory, run the following commands:

```
cmake -D CMAKE_INSTALL_PREFIX=[path to install] \
  -D isc.python_install=Prefix [path to ISC source directory]
cmake --build . --config RelWithDebInfo
ctest --build-config RelWithDebInfo
cmake --install . --config RelWithDebInfo
```

This will configure, build, test, and install the ISC library, utilities, and Python bindings. Testing is optional but recommended. One should confirm all tests pass prior to installation.

The two CMake variables `CMAKE_INSTALL_PREFIX` and `isc.python_install` control where components of ISC are installed. The location for the library and the utilities is controlled by the variable `CMAKE_INSTALL_PREFIX`.

2.3 Building the ISC Python3 Extensions

Building and installing the ISC Python extensions is done by default but can be avoided using CMake by adding the flag `-Disc.python=OFF` to the initial `cmake` command.

The Python binding install location is controlled by `isc.python_install`, which has three options:

- `Global`: This will install in the current global Python module directory, and is most useful for system-wide installs or for Python virtual environments.
- `User`: This will will install in the current user's Python module directory. This is most useful for installing without administration privileges.
- `Prefix`: This will install within `CMAKE_INSTALL_PREFIX/lib`, which is most useful for packaging and maintaining multiple versions. The precise location is OS-dependent, but on Linux, the location will likely be `CMAKE_INSTALL_PREFIX/lib/pythonX.X/site-packages`, where `X.X` corresponds to the specific Python version used to build ISC. In this case, you will have to add the `site-packages` path to the `PYTHONPATH` environment variable for Python to find the bindings. (Default)

Note that ISC will need to be rebuilt and re-installed whenever you upgrade your Python version, e.g., from 3.9.X to 3.10.X.

On Windows, *only* the Visual Studio 2022 build tools have been tested with CMake version 3.23.1 and Python version 3.8 and 3.10. It is assumed that all of the aforementioned applications are in the `%PATH%` in either the user or system environment variables.

Depending on the user's Python installation, it is possible that minor tweaks, e.g., altering some compile or link flags, to the `setup.py.in` file will be required. Builds of the Python bindings have been tested with the Anaconda Python distribution (https://www.anaconda.com/products/distribution) and with the base distribution from https://www.python.org/downloads.

2.4 Installing the Python3 Extensions with `pip`

The ISC release with the MCNP(R)[1] code also ships with "Python Wheel" files to directly install pre-built Python 3.10 bindings. The wheel files can be installed with `pip` using the following command:

```
pip install --prefix /path/to/install/dir isc-X.Y.Z-NNNNNN.whl
```

Above, `/path/to/install/dir` is the location where the ISC package should be installed, and, if it is omitted, defaults to the install location of the Python installation. The `X.Y.Z` is the ISC version number and the `NNNNNN` is a placeholder for information about the system for which the specific wheel file is built, e.g., `win_amd64`.

If the default installation location is not used, then, after the wheel successfully installs, the user will need to ensure that their `PYTHONPATH` points to the installation—for Python version X.X this is typically `/path/to/install/dir/lib/pythonX.X/site-packages`.

3 SZA Identifiers

Because long-lived isomeric states are important when evaluating intrinsic radiation sources, ISC uses an SZA identifier to identify a particular isomer (Conlin et al. 2012). The S refers to the isomeric-state number of the isomer. The isomeric state is **not** necessarily equivalent to the excitation level state as isomeric-state numbers are only assigned to "long-lived" states. The Z refers to the atomic number of the isomer, and the A refers to the mass number of the isomer. The SZA is then formed by the following formulation:

$$\text{SZA} = S \times 1000000 + Z \times 1000 + A \tag{1}$$

[1] MCNP® and Monte Carlo N-Particle® are registered trademarks owned by Triad National Security, LLC, manager and operator of Los Alamos National Laboratory. Any third party use of such registered marks should be properly attributed to Triad National Security, LLC, including the use of the ® designation as appropriate. Any questions regarding licensing, proper use, and/or proper attribution of Triad National Security, LLC marks should be directed to trademarks@lanl.gov.

With the formulation of Eq. 1, the SZA for ^{234m1}Pa (an important isomer in the decay chain of ^{238}U) is 1091235.

4 ISC's Data Sets

The ISC package comes with three different sets of curated data:

1. Natural abundance and mass files—includes natural abundances and masses of isotopes
2. Radioactive decay files—contains half lives, branching ratios, and daughters for decay
3. Particle emission files—contains the emissions from a isotope per radioactive decay

The fundamental data was not produced by the authors, but, rather, has been formatted by the authors into formats to be used with ISC. (This distinction is similar to how NJOY formats nuclear data to be used with MCNP.)

The data files that ship with ISC are in the Extensible Markup Language (XML) format automatically produced by serialization of data objects with the C++ Boost Serialization library. While XML is human readable, the files produced in the serialization process are difficult (but not impossible) to navigate and **are not** intended to be manipulated directly by users. Should users want to alter data in the files, it is recommended that they employ the ISC library outlined in Section 6.

The file structure of ISC's data files is important. All of the ISC utilities look for data in the directory pointed to by the `ISCDATA` environment variable. This environment variable **must** be set on user's systems for the utilities to function. The ISC utilities will expect to find the natural-abundance, radioactive-decay, and particle-emission-index (see Section 4.3) data files in the directory pointed to by the `ISCDATA` environment variable. Subdirectories (named by particle-emission library) of the `ISCDATA` directory contain the particle-emission data for a given SZA.

4.1 Natural Abundance Files

The natural abundance files contain information about isotope masses and natural abundances. These files typically have extensions ending in `.na.xml` in the ISC package. Currently, the only available natural abundance library that ships with ISC is the NIST library (Coursey et al.).

4.2 Radioactive Decay Files

The radioactive decay files contain information about a nuclide's decay processes and include the following information:

1. half-life (and error if available)
2. decay constant (redundant, and error if available)
3. branching ratios for decay paths (and errors if available)
4. daughter isotopes (except for spontaneous fission)

In the ISC package, the radioactive decay data files typically end in the extension `.dk.xml`. Currently, three sets of radioactive decay data are available:

- **`endf6.dk.xml`** radioactive decay data from ENDF/B-VI.8[2]
- **`endf7.dk.xml`** radioactive decay data from ENDF/B-VII.1
- **`endf8.dk.xml`** radioactive decay data from ENDF/B-VIII.0

[2]The ENDF/B-VI.8 data should not be used by most users. This evaluation of radioactive-decay data is known to have errors. It is included with ISC for historical comparisons.

ISC follows the ENDF-6 format integer identifier convention for identifying decay mechanisms. These integer identifiers and the corresponding decay mechanisms are summarized in Table 1.

Table 1. ENDF-6 format radioactive decay identifiers (M. Herman and A. Trkov 2009)

Identifier	Decay Mechanism
1	β^-—beta decay
2	e.c./β^+—electron capture/positron emission
3	isomeric transition
4	α—alpha decay
5	n—neutron emission decay
6	SF—spontaneous fission
7	p—proton emission decay
10	unknown

As indicated in M. Herman and A. Trkov (2009), combinations of decay pathways can be indicated by combining identifiers in a "F.S" format. Here "F" indicates the first decay pathway, and "S" indicates the second decay pathway. Therefor, a decay identifier of 1.5 indicates β^- decay followed by neutron emission.

4.3 Particle Emission Files

The particle-emission files contain the radioactive emissions of a given SZA. The particle-emission files identify the emitted particle types using the ENDF-6 format radioactive particle emission integer types summarized in Table 2.

Table 2. ENDF-6 format radioactive particle emission identifiers (M. Herman and A. Trkov 2009)

Identifier	Radiation Type
0	γ—gamma rays
1	β^-—beta rays
2	e.c./β^+—electron capture/positron emission
4	α—alpha particles
5	n—neutrons
6	SF—spontaneous fission fragments
7	p—protons
8	e^-—discrete electrons
9	x-rays and annihilation radiation

In the ISC package, the particle-emission files typically end in `.{library}.xml` where `{library}` indicates the library that provided the emission data. For example, the ENDF/B-VII.1 radiative emissions from ^{238}U would be found in the file `92238.endf7.xml`. ISC's utilities depend on the particle emission data being located relative to the particle-emission index file. The index file contains the relative path to the particle emission data from the index file. The index files are typically named for the emission library data with a `.idx.xml` extension. For example, the particle-emission index file for the ENDF/B-VII.1 data is `endf7.idx.xml`.

5 Standalone Utilities in the ISC Package

The ISC package comes with two binary utilities. This section describes those utilities. Usage information for all utilities can be obtained by passing the `-h` or `--help` flags.

5.1 ISC's `mattool` Utility

The `mattool` utility was written to facilitate breakdown of natural (S)ZAIDs into their isotopic components. Given a set of (S)ZAIDs and associated fractions (be they atom or mass as specified by the user), `mattool` produces a table of the isotopic and "Z-summed" (natural) atom and mass fractions. Additionally, if a atom or mass density is supplied, then the other (atom or mass) density is also computed.

`mattool`'s usage information is as follows:

```
USAGE: mattool [--version] [--data data] [--natlib natlib] [--atomfracs]
       [--massfracs] [--atomden atomden] [--massden massden]
       <ZAID-Fractions [ZAID-Fractions ... ]>

DESCRIPTION:

mattool takes ZAID and atom/mass fraction information and produces material
specification information

OPTIONS:

--version, -v     : Print version information and exit

--data, -d        : Set natural abundance library (default: nist.na.xml)

--atomfracs, -a   : Specified fractions are atom fractions

--massfracs, -m   : Specified fractions are mass fractions

--atomden         : Specify atom density

--massden         : Specify mass density

ZAID-Fractions    : ZAID-fraction pairs
```

As an example, consider one wants to determine what the isotopic constituents of water (H_2O) are. The `mattool` execution line that would provide this information is

```
mattool -a 1000 2 8000 1
```

where the `-a` flag indicates that the `2` and `1` are the **atom** fractions (`mattool` will internally normalize the values) of hydrogen (1000) and oxygen (8000), respectively. The output produced by this command follows:

```
  zaid      atom frac     z atom sum      mass frac     z mass sum
========  =============  =============  =============  =============
    1001   6.665900e-01                  1.118727e-01
    1002   7.666667e-05   6.666667e-01   2.571391e-05   1.118984e-01
    8016   3.325233e-01                  8.856949e-01
    8017   1.266667e-04                  3.585660e-04
    8018   6.833333e-04   3.333333e-01   2.048165e-03   8.881016e-01
```

In the above output one notes that the atom and mass fractions of the individual isotopes are produced along with their sums over atomic number (i.e., z atom/mass sum). Additionally, the mass (or atom) density could have been specified on the command line as follows:

```
mattool -a 1000 2 8000 1  --massden 1.0
```

Adding the density produces the following *addtional* lines of output:

```
Mass Density: 1.000000e+00
Atom Density: 1.002839e-01
```

By default, `mattool` will used the isotopic natural abundances in the NIST database of natural abundances (i.e., the `nist.na.xml` data file located in the directory at which the `ISCDATA` environment variable points). While no other isotopic natural abundance files are currently provided, one could override the default by specifying the full path to an alternate data file on the `--data/-d` flag.

5.2 ISC's MCNP Intrinsic Source Constructor (`misc`) Utility

The `misc` utility is a standalone application to generate MCNP[3] `SDEF` distributions. The `misc` utility reads an input file and produces an output file with a summary of the calculation performed and a source file containing an `SDEF` distribution that can be copied into an MCNP input or included in an MCNP input with the `READ` card.

The usage of the `misc` utility is as follows:

```
USAGE: ./utils/misc/misc [--version] [infile]

DESCRIPTION:

./utils/misc/misc generates MCNP SDEF descriptions for radioactive material
descriptions

OPTIONS:

--version, -v     : Print version information and exit

infile            : MISC input file
```

Currently, all input for the `misc` utility is provided via the input file. Available inputs arguments are documented in the MISC User Guide (C.J. Solomon 2012).

6 The ISC Library

This section will describe *some* of the many classes that are part of the ISC code package. A full description at this juncture is not merited because the ISC class implementations are likely to change in the near future. Additionally, the class methods documented herein are generally the "getter" methods (users interested in changing data within data files are referred to the class header files for the "setter" methods). As mentioned in the installation section, the ISC package is written in C++ and bound to Python, so most of the example code will be presented in Python.

6.1 Classes for Managing Data Files

The ISC data files are managed through 4 classes, the "File" classes, that provide functionality for reading and writing, but not accessing, the respective data. Access to data is provided through 3 additional classes, the "Library" classes. For example, the `DecayFile` class can read and write radioactive decay data, but

[3]MCNP® and Monte Carlo N-Particle® are registered trademarks owned by Triad National Security, LLC, manager and operator of Los Alamos National Laboratory. Any third party use of such registered marks should be properly attributed to Triad National Security, LLC, including the use of the ® designation as appropriate. Any questions regarding licensing, proper use, and/or proper attribution of Triad National Security, LLC marks should be directed to trademarks@lanl.gov.

the `DecayLib` class is used to access the data and is typically constructed by passing it an instance of a `DecayFile` class.

6.1.1 The `AbundanceFile` and `AbundanceLib` Classes

The `AbundanceFile` class has only two public methods of interest that are summarized in the following table:

Method	Description
`AbundanceFile(FILE,TYPE)`	construct `AbundanceFile` by passing file name FILE and TYPE (defaults to XML).
`Insert(SZA,MASS,ABUND)`	insert an entry into the data file for SZA with mass MASS and abundance ABUND

The `AbundanceLib` has the following public member methods of interest

Method	Description
`AbundanceLib(ABUNDFILE)`	construct an abundance library object by passing it an instance of an `AbundanceFile` object
`GetMass(SZA)`	return the mass for the given SZA
`GetAbundance(SZA)`	return the abundance for a given SZA
`GetZs()`	return all atomic numbers with data in the `AbundanceLib`
`GetIsosForZ(Z)`	return all the SZAs that exist in the `AbundanceLib` provided the given Z
`HasNaturals(Z)`	return `true` if the `AbundanceLib` has natural abundances for SZAs with the given Z

The following example illustrates how to read abundance file data into an `AbundanceFile` class, construct an `AbundanceLib` class from the abundance file, and query the naturally occurring SZAs of uranium.

```
1  import isc # import the isc module
2
3  # open an abundance data file and convert it to an abundance library
4  abund_file = isc.AbundanceFile(os.path.join(iscdata,"nist.na.xml"))
5  abund_lib = isc.AbundanceLib(abund_file)
6
7  # get all U isotopes with data
8  u_isos = abund_lib.GetIsosForZ(92)
9  print(u_isos)
10
11 # get mass and abundance for each naturally occuring isotope
12 nat_u_isos = list()
13 for iso in u_isos:
14   # get mass and abundance for iso
15   mass = abund_lib.GetMass(iso)
16   abundance = abund_lib.GetAbundance(iso)
17   print("{:7d}  {:7.3f}  {:12.5e}".format(iso, mass, abundance))
18
19   # if abundances is non-zero add it to the list of naturally occurring isos
20   if( abundance > 0.0 ):
```

```
    nat_u_isos.append(iso)
print(nat_u_isos)
```

6.1.2 The DecayData, DecayFile, and DecayLib Classes

The DecayData class contains information about the decay mechanisms of a given SZA. The useful public methods of the DecayData class are as follows:

Method	Description
GetZ()	return atomic number of isomer for which this decay data applies
GetA()	return mass number of isomer for which this decay data applies
GetS()	return isomeric state number of isomer for which this decay data applies
GetHalfLife()	return the half life of this isomer in *seconds*
GetHalfLifeErr()	return the uncertainty in the half life in *seconds*
GetDecayConst()	return the radioactive decay constant in *1/seconds*
GetDecayConstErr()	return the uncertainty in the radioactive decay constant in *1/seconds*
GetNumber()	return the number of decay pathways
GetBranchingRatio(N)	return the branching ratio of the Nth decay pathway
GetBranchingRatioErr(N)	return the uncertainty in the branching ratio of the Nth decay pathway
GetDaughter(N)	return the daughter of the Nth decay pathway
GetDecayType(N)	returns the decay type (see Table 1) of the Nth decay pathway

The DecayFile class has the following public methods to read data from a file and insert data:

Method	Description
DecayFile(FILE,TYPE)	constructs a DecayFile class from file name FILE having type TYPE (typically XML)
SetDecayData(SZA, DATA)	set the radioactive decay data for SZA to a DecayData instance DATA

The DecayLib class has the following public methods to access data:

Method	Description
DecayLib(DECAYFILE)	construct a DecayLib class by passing it an instance of a DecayFile class DECAYFILE
GetSZAs()	return a list of SZAs for which there is data in the DecayLib class
GetDecayData(SZA)	return a DecayData class of the decay data for the given SZA
GetAllDaughers(SZA)	return a list of all the daughters of a given SZA

The following example illustrates how to open decay data from a file into the DecayFile class, convert the DecayFile into a DecayLib, and query the DecayData for ^{137}Cs.

```
import isc # import the isc module

# open a decay data file and convert it to a decay data library
decay_file = isc.DecayFile(os.path.join(iscdata,"endf7.dk.xml"))
decay_lib = isc.DecayLib(decay_file)

```

```
# get all the daughters of Cs-137
cs137_daughters = decay_lib.GetAllDaughters(55137)
print(cs137_daughters)

# get the decay data for Cs-137
cs137_decay_data = decay_lib.GetDecayData(55137)
print("Cs-137 half life = {:12.5e} s".format(cs137_decay_data.GetHalfLife()))

# loop over the number of decay pathways
for i in range(cs137_decay_data.GetNumber()):
  # get daughter SZA is branching ratio
  daughter = cs137_decay_data.GetDaughter(i)
  branching_ratio = cs137_decay_data.GetBranchingRatio(i)
  print("{:7d}  {:12.5e}".format( daughter, branching_ratio ) )
```

NOTE: Many similar packages to ISC have treated decay of ^{137}Cs as where the 662 keV emission comes directly from ^{137}Cs. In reality, the 662 keV emission comes from ^{137}Cs's daughter ^{137m}Ba. ISC treats these daughters explicitly without assumption regarding secular equilibrium.

6.1.3 The `EmissionFileIndex`, `EmissionFile`, and `EmissionLib` Classes

The `EmissionFileIndex` class has the following public methods for users:

Method	Description
`EmissionFileIndex(FILE,TYPE)`	construct the `EmissionFileIndex` by passing in file name FILE and type TYPE (typcially XML)
`GetPath(SZA)`	return the path to the particle-emission data for SZA
`HasPath(SZA)`	return `true` if the given SZA has a particle-emission file

The public methods of the `EmissionFile` class are the following:

Method	Description
`GetZ()`	return atomic number of isomer for which this emission data applies
`GetA()`	return mass number of isomer for which this emission data applies
`GetS()`	return isomeric state number of isomer for which this emission data applies
`GetDiscreteTypes()`	return a list of all particle types (see Table 2) for which there are discrete emissions
`GetContinuumTypes()`	return a list of all particle types (see Table 2) for which there are continuum emissions
`GetWattSpectrumTypes()`	return a list of all particle types (see Table 2) for which there are Watt spectrum emissions
`GetDiscreteNumber(PT)`	return the number of discrete emissions for particle type PT
`GetDiscrete(PT,N)`	return the `N`th discrete emission for particle type PT
`GetContinuumNumber(PT)`	return the number of continuum emissions for particle type PT
`GetContinuum(PT,N)`	return the `N`th continuum emission for particle type PT
`GetWattSpectrum(PT)`	return the Watt spectrum for particle type PT

The `EmisionLib` class has the following public methods:

Method	Description
`SetFromEmissionFile(FILE)`	set the emissions for an SZA from the emission file FILE
`GetSZAs()`	return a list of SZAs in the emission library
`GetSpectra(SZA)`	return the emission spectra for the given SZA

The following example demonstrates how to open emission files found in an `EmissionFileIndex` into `EmissionFile` classes, add the `EmissionFile` data to an `EmissionLib`, and query data out of the `EmissionLib` for ^{60}Co.

```
# open an emission file index (contains relative paths to emission data files)
emission_index = isc.EmissionFileIndex(os.path.join(iscdata,"endf7.idx.xml"))
# initialize an empty emission library
emission_lib = isc.EmissionLib()

# loop over all SZAs and import the emission data
# NOTE: one need not load everything, only the things you need
for sza in emission_index.GetSZAs():
  print("loading emission data for isotope {:d}".format(sza))
  emission_file = isc.EmissionFile( os.path.join(iscdata,emission_index.GetPath(sza)) )
  emission_lib.SetFromEmissionFile( emission_file )

# get the emission spectra for Co-60
co60_spectra = emission_lib.GetSpectra(27060)

# get a list of isc particle types for which spectra exist
co60_particle_types = co60_spectra.GetParticleTypes()
print(co60_particle_types)

# loop over the particle types
for ptype in co60_particle_types:
  if ptype == isc.ENDF_DECAY_GAMMA:
    print("Co-60 emits gammas, ptype = {:d}".format(ptype))
  elif ptype == isc.ENDF_DECAY_BETAM:
    print("Co-60 emits beta-, ptype = {:d}".format(ptype))
  elif ptype == isc.ENDF_DECAY_BETAP:
    print("Co-60 emits beta+, ptype = {:d}".format(ptype))
  elif ptype == isc.ENDF_DECAY_IT:
    print("Co-60 has internal transition, ptype = {:d}".format(ptype))
  elif ptype == isc.ENDF_DECAY_ALPHA:
    print("Co-60 emits alphas, ptype = {:d}".format(ptype))
  elif ptype == isc.ENDF_DECAY_ELECTRON:
    print("Co-60 emits electrons, ptype = {:d}".format(ptype))
  elif ptype == isc.ENDF_DECAY_XRAY:
    print("Co-60 emits xrays, ptype = {:d}".format(ptype))

# get the gamma spectrum
co60_gammas = co60_spectra.GetSpectrum(isc.ENDF_DECAY_GAMMA)
print("The number of discrete emissions per decay is {:.3f}".format(\
  co60_gammas.GetDNorm()))

# loop over all the discrete emissions
```

```
print("{:12s}  {:12s}".format("energy","#/decay"))
for i in range(co60_gammas.GetDNumber()):
  # get the emission energy and probability/decay
  energy = co60_gammas.GetDEnergy(i)
  intensity = co60_gammas.GetDIntensity(i)
  print("{:12.5e}  {:12.5e}".format(energy,intensity))
```

6.2 The `DecayMaterial` Class

The `DecayMaterial` class is the primary class used to construct a radioactive material and build its emissions. To construct a `DecayMaterial` one must pass the constructor an `AbundanceLib` (so that natural isotopes can be expanded, a list of SZAs, their corresponding fractions, a flag for whether the fractions are atom or mass fractions, a density, and a flag for whether or not the density is an atom or mass density. The `DecayMaterial` class has the following public methods:

Method	Description
`Age(ABUNDLIB, DECAYLIB, TIME)`	age the `DecayMaterial` using `AbundanceLib` ABUNDLIB and `DecayLib` DECAYLIB for at time TIME in *seconds*
`BuildSource(DLIB, ELIB, EBREMS)`	build the source description using data from `DecayLib` DLIB and `EmissionLib` ELIB; if EBREMS is `true`, then electron emissions are converted into bremsstrahlung emissions using a thick-target bremsstrahlung model
`Reset()`	reset the material zaids and fractions to the initial specification (i.e., that before any aging has been applied)
`GetSZAs()`	return the list of SZAs in the material; if aging has been performed this will include all daughters
`GetAtomFrac(SZA)`	return the atom fraction of the given SZA
`GetMassFrac(SZA)`	return the mass fraction of the given SZA
`GetAtomDensity()`	return the atom density of the `DecayMaterial`
`GetMassDensity()`	return the mass density of the `DecayMaterial`
`GetSpectra()`	return an `EmissionSpectra` class containing emission spectra of all particle types (see Table 2) from all SZAs
`GetSpectra(SZA)`	return an `EmissionSpectra` class containing emission spectra of all particle types (see Table 2) from the given SZA

The following example demonstrates how to construct a `DecayMaterial` for natural uranium, age it for 1 year, and obtain the emission spectra (for brevity it is assumed that `AbundanceLib al`, `DecayLib dl`, and `EmissionLib el` are available):

```
# build the DecayMaterial using
#   - the AbundanceLib to expand natural SZAs
#   - the isc.DecayMaterial.ATOM flag to specify atom fractions
#   - a density of 19
#   - the isc.DecayMaterial.MASS flag to specify mass density
natu = isc.DecayMaterial(al, [92000], [1.0], isc.DecayMaterial.ATOM, 19.0, isc.DecayMaterial.MASS)

# age the natural uranium for 1 yr = 365.24 * 24 * 3600 = 31556736.0 s using specified
# AbundanceLib and DecayLib
natu.Age(al, dl, 31556736.0)

# build the source using specified DecayLib and EmissionLib
```

```
natu.BuildSource(dl, el)

# obtain the emission sepctra
spectra = natu.GetSpectra()

# reset the material to the un-aged state
natu.Reset()
```

6.3 The EmissionSpectra and EmissionSpectrum Classes

The `EmissionSpectra` class is a collection of particle types (see Table 2) with their corresponding `EmissionsSpectrum`. An `EmissionSpectrum` consists of two pieces: 1. discrete emissions and 2. continuum emissions. `EmissionSpectra` are most commonly obtained from calls to the `GetSpectra()` method of the `DecayMaterial` class; in this case, all emission intensities are in units of $\#/\mathrm{cm}^3/\mathrm{s}$.

The public methods of the `EmissionSpectra` class are the following:

Method	Description
`GetParticleTypes()`	return a list of particle types (see Table 2) that have emissions
`HasSpectrum(PT)`	return true if the particle type `PT` (see Table 2) has a spectrum
`GetSpectrum(PT)`	return the emission spectrum for particle type `PT` (see Table 2)
`ToMCNPTypes(BREMS2PHOT)`	return an `EmissionSpectra` class where the particle types are mapped into MCNP particle types; if `BREMS2PHOT` is `true`, the bremsstrahlung spectrum is added to the photon spectrum
`Clear()`	clear the spectra

The `EmissionSpectrum` class has the following public methods:

Method	Description
`GetDNumber()`	return the number of discrete emissions in the spectrum
`GetDEnergy(N)`	return the energy of the `N`th discrete emission
`GetDEnergyErr(N)`	return the uncertainty in the energy of the `N`th discrete emission
`GetDIntensity(N)`	return the intensity of the `N`th discrete emission
`GetDIntensityErr(N)`	return the uncertainty in the intensity of the `N`th discrete emission
`GetDDecayType(N)`	return the decay type (see Table 1) of the `N`th discrete emission
`GetCNumber()`	return the number of continuum emissions bins
`GetCEnergy(N)`	return the energy of the `N`th continuum emission bin
`GetCIntensity(N)`	return the intensity of the `N`th continuum emission bin
`GetDNorm()`	return the total intensity of all discrete emissions
`GetCNorm()`	return the total intensity of all continuum emissions

Continuing from the code listing example in Section 6.2 (assuming that the `Reset()` method wasn't called) the photon emission spectrum from natural uranium can be obtained as follows:

```
# obtain the emission spectra
spectra = natu.GetSpectra()

# gamma emission spectrum
gammas = spectra.GetSpectrum(isc.ENDF_DECAY_GAMMA)

```

```
# iterate over the discrete gamma emissions and print energies and intensities
# if the intensity is greater than 1/10000th of the total intensity
for i in range(gammas.GetDNumber()):
    energy = gammas.GetDEnergy(i)
    intensity = gammas.GetDIntensity(i)
    if intensity > gammas.GetDNorm() / 10000:
        print("{:12.5e}  {:12.5e}".format(energy, intensity))
```

7 Putting it All Together

The following example illustrates how one could build a photon emission source for natural U, using the internal thick-target bremsstrahlung model to convert electron emissions into photons.

```
# This file uses ISC to generate photon emissions off of natural uranium

import os

import isc # import the ISC package

# assume the ISCDATA path variable is set to the ISC data directory
iscdata = os.getenv("ISCDATA")

# open abundance file and create abundance library
af = isc.AbundanceFile( os.path.join(iscdata, "nist.na.xml") )
al = isc.AbundanceLib( af )

# open decay file and create decay library
df = isc.DecayFile( os.path.join(iscdata, "endf7.dk.xml") )
dl = isc.DecayLib( df )

# open the emission file index and load all emission files into the
# emission library
el = isc.EmissionLib()
eidx = isc.EmissionFileIndex( os.path.join(iscdata,"endf7.idx.xml") )
for sza in eidx.GetSZAs():
    ef = isc.EmissionFile( os.path.join(iscdata,eidx.GetPath(sza)) )
    el.SetFromEmissionFile( ef )

# create the natural uranium material at a density of 18.9 g/cc the natural SZA
# 92000 will be expaned automatically with the abundance library
natu = isc.DecayMaterial(al, [92000], [1.0], isc.DecayMaterial.ATOM, 18.9, isc.DecayMaterial.MASS)

# Age the material for 1 year to build in daughter
natu.Age(al, dl, 365.24 * 24 * 3600)

# Build the source spectrum
natu.BuildSource(dl, el, True) # covert electron-emission to bremsstrahlung

# get the gamma emissions
gammas = natu.GetSpectra().GetSpectrum(isc.ENDF_DECAY_GAMMA)

```

```
# print the discrete emission
for i in range(gammas.GetDNumber()):
    print("{:12.5e}  {:12.5e}".format(gammas.GetDEnergy(i), gammas.GetDIntensity(i)))

# get the bremsstrahlung
brems = natu.GetSpectra().GetSpectrum(isc.ENDF_DECAY_BREMS)

# print the bremsstrahlung continuum
for i in range(brems.GetCNumber()):
    print("{:12.5e}  {:12.5e}".format(brems.GetCEnergy(i), brems.GetCIntensity(i)))
```

References

C.J. Solomon. 2012. "MCNP Intrinsic Source Constructor (MISC): A User's Guide." Report LA-UR-12-20252. Los Alamos National Laboratory.

Conlin, J. L., F. B. Brown, A. C. Kahler, M. B. Lee, D. K. Parsons, and M. C. White. 2012. "Version 2.0.0 of ACE Tables Header Format." Report LA-UR-12-25177. Los Alamos National Laboratory.

Coursey, J. S., D. J. Schwab, J. J. Tsai, and R. A. Dragoset. "Atomic Weights and Isotopic Compositions with Relative Atomic Masses." NIST Physical Measurement Laboratory. https://www.nist.gov/pml/atomic-weights-and-isotopic-compositions-relative-atomic-masses.

M. Herman and A. Trkov, ed. 2009. *ENDF-6 Formats Manual*.