Comments on an Infrastructure for GND

Bret Beck

Presented at the OECD/NEA/WPEC SG38

Tokai, Japan

10-Dec-2013

Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94551

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and partly funded by the Nuclear Data Program Initiative of the American Recovery and Reinvestment Act (ARRA).

LLNL-PRES-

Statement of work

This team will develop a list of agreed-upon infrastructure functions and specify their name, purpose, inputs and outputs in order to facilitate infrastructure development and testing. These teams will also begin the process of releasing their infrastructure codes as open source as their individual institutions permit, possibly as partial releases if necessary. The various SG38 contributors will update their infrastructure to work with GND structures.

Infrastructure outline

- Initial comments
  - What language type?
  - Only need to support GND/XML?
  - Unit conversion
  - Reaction matching – searching
- Purpose of the infrastructure
- Some needed methods/Function
- Issues from Fudge work
  - Floating point
  - Keeping it clean while supporting slobs
  - Adding instance to structure should copy

Initial comments

- Structures, and infrastructure/API are entwined
  - Care must be taken when designing the structure so as not to
    - Make the infrastructure/API to complex
    - Make the infrastructure/API consume too much computational
      - memory
      - time

- Using the infrastructure to modify a file is preferred to using cut and paste with an editor

- Indexing should be 0 based
  - Note, in Fortran can define arrays as "(0:n-1)"
    - Example: real x(0:n-1)

Type of language for user interaction

- I believe infrastructure should be written in an Object Orientated Programming (OOP) language, unlike API.
  - e.g., C++, Java, Python
  - Why
    - It makes the naming simpler and language more intuitive
      - Consider plotting some data
        - » In OOP, with well designed classes, this is "data.plot( )" for all data types
        - » In other language, each data type needs a different function and the user needs to know the type of data to call the proper function
          - plotXYData(dataXY), plotXYZData(dataXYZ), etc.
    - Iterators make looping easier

Infrastructure only needs to read/write GND/XML?

- Infrastructure codes only need to support reading/writing to XML
  - Format = Structure + meta-language
    - Written as “Structure/meta-language”
    - Examples: “GND/XML”, “GND/HDF5”
  - Conversion to other formats can be done with codes that only need to know a little of the GND structure as long as the structure is well defined.
    - Example: In fudge, toHDF5.py is a stand alone python script that converts a “GND/XML” file into “GND/HDF5”. toHDF5.py is roughly 110 lines of python and does not import anything from Fudge.
- Conversion to non-GND formats discussed later
  - e.g., ACE files used as input to MCNP

Unit conversion

- ENDF stores data as energy in eV and cross section in barn
- ACE files stores energy in MeV
- I believe that the infrastructure should support unit conversion of the data.

```
GND_instance.changeUnitsTo( { “energy” : “MeV”, “crossSection” : “mb” } )
```

- What is involved.
  - New structure must store units
  - Data must indicate data type for each axis
    - $\sigma(E)$ – E axis is energy, σ axis is cross section
    - $P(E'|E)$ – E and E’ axes are energy, P axis is 1/energy
    - GND does not store axis data this way – I am planning to change GND and Fudge to support this so that others can see an example

Reaction matching

- Need to have a 'regular expression' like pattern matching for reactions
  - Useful for matching in loops and searching
  - 'Regular expression' example
    - "n+(H1)?" - match one or more 'n' followed by 0 or 1 'H1'.
    - "n{2,3}(H1)?.*" – must contain 2 or 3 'n's followed by 0 or 1 'H1' and then anything else (i.e., '.*'). Here '.*' should probably on match a valid particle
  - Some things to consider
    - Order should and should not matter – user should be able to specify
      'n + p' versus 'p + n'
    - Decay should and should not matter
      'n + p + H2' versus 'n + (He3 → p + H2)' or
      'p + n + H2' versus 'p + (H3 → n + H2)'
    - Search by product energy (e.g., gamma with outgoing energy more than 6 MeV)
      - How do we make a "regular expression" for matching this?

Reaction matching - continue

- Some more things to consider
  - Should be able to match or not the following reactions

H3 + H3 → 2n + He4
n + (He5 → n + He4)
n + (He5_e1 → n + He4)

Purpose of the infrastructure

Main purpose of the infrastructure

- Allow people to easily
  - read/write data
  - plot data
  - compare data
    - Useful for comparing processed data against output from another infrastructure
  - process data
    - Reconstruct resonances
    - Calculate average energy and momentum to products
    - Heat cross sections
    - Deterministic grouping
    - Others: (e.g., URR probability tables)
  - Remove/add reactions/data
    - helpful when testing/comparing processing codes
  - All of the above need a way to select parts of the structure
    - i.e., reaction/data matching

Some needed methods/Function

Read/write XML methods

- readXML(fileName)

- saveAsXML(fileName)

Unit conversion

- How I would define method if only using python
  - changeUnitsTo({ “energy” : “MeV”, “crossSection” : “mb” })
    - Uses the python dictionary which is a list of key/value pair
    - Use string for keys and values

Key	Value
“energy”	“MeV”
“crossSection”	“mb”

- General solution for any OOP language?
  - associative array
    - C++ std::map, Java Map, Python dict.
    - Keys and values are strings
  - changeUnitsTo(associativeArray)

Linearizing data

- toLinear – at least for the last two columns for each data type
  - $\sigma(E)$, for both E and σ
  - $P(\mu|E)$, only for μ and P
  - $P(\mu,E'|E)$, only for μ and P
  - $P(E',\mu|E)$, only for E' and P
  - Including parametric data (e.g., Reconstruct resonances)
  - Do not have to switch frames (too hard)!
    - Except for deterministic processing
  - Possible arguments:
    - accuracy, lowerEps, upperEps

- Currently in Fudge most toLinear methods are called
  - toPointwise_withLinearXYs(accuracy, lowerEps, upperEps)

Plotting

- Plot( )
  - Possible arguments:
    - linear or log per axis
    - min, max per axis
    - title
  - Labels and units come from data (e.g., axes information)
    - a lesson I learned while developing Fudge
- Multiple curves on one plot (as Grucon allows)
- Only need to support plotting for pointwise data
- What about functional forms, do we want to have a way to plot parameters?

Search/matching reactions

- In addition to reactions, we probably want other data types (e.g., cross section), products

- ?

Processing

- Sn grouping (i.e., deterministic grouping)
  - Inputs:
    - Required: GND/XML file, group boundaries, flux, Temperature
    - Optional: verbosity, logFile, workDirectory, Tolerance (rel. and abs.) other?

- Heating cross sections
  - Argument for Fudge heating method
    - temperature, EMin, lowerlimit , upperlimit , interpolationAccuracy, heatAllPoints, doNotThin, heatBelowThreshold, heatAllEDomain

- Reconstruct resonances
  - Optional: Tolerance (rel. and abs.)

- Calculate average outgoing energy and momentum to products
  - Useful for check methods.

- Others?

General useful methods/functionality we may want to consider

- These are currently in Fudge
  - Iterate over objects that are list-like or dictionary like
  - Copy for many classes
  - Methods to remove/add parts of structure
  - Get data domain
    - Possible arguments: data.getDomain(inUnitOf = “MeV”)
  - Check - verify data
  - Data thinning

Some issues

Floating point precision

- Is ‘a – a’ zero?
- Not always with finite floating-point precision

```
>>> 1e-5 * 1e-6 / 1e-11 - 1
2.220446e-16
```

- We probably need to define fuzzy comparisons
  - For example, a and b are equal if

$$\text{abs}(a - b) \leq \text{epsilon} * \max(\text{abs}(a), \text{abs}(b))$$

  - Example python ‘compare’ method

```
def compare( value1, value2, epsilon = 0 ) :

    epsilon *= sys.float_info.epsilon
    delta = value1 – value2
    if( abs( delta ) <= epsilon * max( abs( value1 ), abs( value2 ) ) ) : return( 0 )
    if( delta < 0. ) : return( -1 )
    return( 1 )
```

Outputted independent data should be unique

- As example, for σ(E) data the independent data are the E values
  - The E values should be written with enough digits so they are unique.

Adding instance (object) to another instance should make a copy

- Example from python

```
pw_multiplcity = [ [ 1e-5, 2.2 ], [ 20, 3.2 ] ]
multiplicity = Multiplicity.pointwise( pw_multiplcity )
pw_multiplcity[1][1] = 4.2
```

  - Should the third line change the value stored in multiplicity
    - In python, it will if a copy is not made and will not if a copy is made
  - In Fudge, we have re-written many methods to copy data added to them.
    - Now you know what my opinion is!

The end

What is currently in Fudge

- Written in python using classes
- Top level class is called reactionSuite
  - Contains all reactions for a projectile hitting a target
- Simple to loop over all reactions
  - For each reaction instance
    - Cross section component is instance.crossSection
      - Has method toPointwise_withLinearXYs
    - Products are in instance.outputChannel

Some of the classes in Fudge for handling data

Axes classes : for storing interpolations, labels and units
- interpolationXY
- axis, interpolationAxis
- axes, referenceAxes, interpolationAxes

Class for storing a list of floats
- Ys

Classes for storing data of the form y(x) (e.g., $\sigma(E)$)
- XYs, regionsXYs, XYs_LegendreSeries

Classes for storing data of the form y(x|w) (e.g., $P(\mu|E)$)
- W_XYs, W_XYs_LegendreSeries
  - W_XYs is a list of XYs with a w-value and interpolation.

Classes for storing data of the form y(x|w,v) (e.g., $P(\mu|E',E)$)
- V_W_XYs, V_W_XYs_LegendreSeries
  - V_W_XYs is a list of W_XYs with a v-value and interpolation.

Class for storing a polynomial
- polynomial

Processing and saving Methods

- Would like infrastructure to make it easy for others to process data for their needs and save to their format
  - Example: save versus saveAs
    - Save is to XML
    - saveAs: ACE (LANL MC), NDI (LANL Sn), ndf (LLNL Sn)

- Processing:
  - My current opinion is to have LLNL's Sn processing in Fudge
  - Others can add their own processing methods (processAs) via class inheritance

- Saving
  - Default save should be to XML.

Class inheritance option for processAs and saveAs

- Example of SaveAsACE
  - Items in Fudge

```
LICENSE.txt                          MANIFEST.in      Makefile      Misc
README.txt                           Testing/         bin/          build/
crossSectionAdjustForHeatedTarget/                    doc/          examples
fudge/                               numericalFunctions/  pqu/      setup.py
siteSpecific/
```

  - Example "siteSpecific/" directory

```
BNL/    KAPL/    LANL/    LLNL/
```

    - Note, KAPL may not release is code, but when they download the latest Fudge, they can copy their KAPL directory into siteSpecific/

  - Example of LANL/

```
saveAsACE.py    saveAsNDI.py
```

Sketch of saveAsACE.py – another good reason for OO

```
from fudge.gnd.reactionSuite import reactionSuite as baseReactionSuite
from fudge.gnd.reactions.reaction.reaction as baseReaction
from fudge.gnd.reactionData.crossSection import component as baseCrossSection

class reactionSuite( baseReactionSuite ) :
    def saveAsACE( self, … ) :
        …
        self.reconstructResonances( … )
        for reaction in self : reaction.saveAsACE( … )
        …

class reaction( baseReaction ) :
    def saveAsACE( self, … ) :
        …
        self.crossSection.saveAsACE( … )
        …

class crossSection( baseCrossSection ) :
    def saveAsACE( self, … ) :
        xSec = self. toPointwise_withLinearXYs( )
        …
```

Thanks to Paul Romano of KAPL for this idea

Possible ways to indicate data type - Currently in GND

- Each dataset (e.g., cross section) has an “axes” element that consist of “axis” elements.
- Each axis element has a label, unit and if independent axis interpolation attributes
- Disadvantage
  - Definitions of similar axes, axis types occur in many places (redundant information)
- Example of pointwise cross section from GND
  - Redundant as it occurs in each reaction

```
<pointwise xData=“XYs”>
    <axes>
        <axis index=“0” label=“energy_in” unit=“eV” interpolation=“linear,linear”/>
        <axis index=“0” label=“crossSection” unit=“b”/></axes>
    <data> 1e-5 1 1e-4 2 … 2e7 6.5</data></pointwise>
```

Possible ways to indicate data type – Better?

- Have an “axes template” element that contains an “axes” template element for each type of data (e.g., cross section, pointwise angular, multiplicity)
- Each dataset would have an axes that references the appropriate “axes template” and can override the interpolation attribute
- Disadvantage:
  - Definition of a unit type (e.g., projectiles energy) will still occur many times (redundant information)

```
<axesTemplates>
    <axes label="crossSection">
        <axis index="0" label="energy_in" unit="eV" interpolation="linear,linear"/>
        <axis index="0" label="crossSection" unit="b"/></axes></axesTemplates>
…
    <pointwise xData="XYs">
        <axes template="crossSection"/>              # Probably should be a link
        <data> 1e-5 1 1e-4 2 … 2e7 6.5</data></pointwise>
…
    <pointwise xData="XYs">
        <axes template="crossSection"><axis index="0" interpolation="linear,log"/></axes>
        <data> 1e-5 1 1e-4 2 … 2e7 6.5</data></pointwise>
```

Possible ways to indicate data type – Best?

- Have a “units” element
  - Contains “unit” elements
  - Each unit element defines the unit for a given axis type
- Have an “axes template” element
  - contains an “axes” template element for each type of data (e.g., cross section, pointwise angular, multiplicity)
    - Contains an “axis” element for each axis of the data type
    - Correlates axis “label” with “unit” with same label in “units” element to determine unit (e.g., “eV”, “b”)
- Each dataset would have an axes that reference the appropriate “axes template” and can override the interpolation attribute
- Advantage:
  - Unit only defined in one spot for each data axis type (no redundant information)

Possible ways to indicate data type – Best? - example

```
<axesTemplates>
    <units>
        <unit label="energy_in" value="eV" type="energy"/>
        <unit label="energy_out" link="../energy_in"/>
        <unit label="crossSection" value="b" type="crossSection"/>
        <unit label="mu" value=""/>
        <unit label="P(mu|energy_in)" value="1/${energy_out}"/></units>
    <axes label="crossSection">
        <axis index="0" label="energy_in" interpolation="linear,linear"/>
        <axis index="1" label="crossSection"/></axes>
    <axes label="P(mu|energy_in)">
        <axis index="0" label="energy_in" interpolation="linear,linear"/>
        <axis index="1" label="mu" interpolation="linear,linear"/>
        <axis index="2" label="P(mu|energy_in)"/></axes></axesTemplates>
…
    <pointwise>
        <axes template="crossSection"><axis index="0" interpolation="linear,log"/></axes>
        <data> 1e-5 1 1e-4 2 … 2e7 6.5</data></pointwise>
…
    <pointwise>
        <axes template="P(mu|energy_in)"/>
        <data> 1e-5 1 1e-4 2 … 2e7 6.5</data></pointwise>
```

GND_instance.changeUnitsTo({ "energy" : "MeV", "crossSection" : "mb" })