Proposing a JSON structure for calculation results

W. Haeck

May 19-24, 2019

May 19-24, 2019

Beijing, China

- **Introduction**
- **Overview**
- **Python coding and examples**

Introduction

Every calculation code gives its result in its own format

- MCNP (LANL): the output file, the mctal file, the ptrac file, etc.
- PARTISN (LANL): the output file, etc.
- MORET (IRSN): the output file, an XML file, etc.

The results we're interested in are however the same

- Single values: keff, βeff, etc.
- Histogram data: particle spectra, reaction rates, sensitivity profiles, etc.
- Pointwise data: nuclide composition as a function of time, etc.

General calculation results

A calculation result consists of two distinct components

- Attributes (or metadata) that give information about the result
  - What type of result is it?
  - What nuclide and reaction is it for (if it is a reaction rate)?
  - What volume is it for (if it is a particle spectrum)?
  - Which code and version produced the result?
  - Which nuclear data library was used to produce the result?
  - This is what we will want to search and filter on
- The actual calculation result
  - Values for the result: a single value, an array of values, an array of arrays of values, etc.
  - Optional uncertainties: in the same form as the values for the result
  - The structure of the result: a histogram of flux values as a function of incident energy, etc.
  - Optional units for the values and uncertainties
  - This is what we will want to compare, store, exchange, plot, etc.

General calculation results

There are some requirements to a structure to store these

- A result should stand by itself
  - I don't need to look in multiple places to be able to understand it
  - For example: for a particle spectrum, I should have the group structure at the same time, etc.
- It should be calculation code agnostic
  - I don't need to know where it came from to understand it
- It should be result type agnostic
  - I can use it for any result, even the ones I have not thought of yet
- It should be relatively easy to interact with through different scripting or programming languages
  - I can use it in python or C++, etc.
- It should be relatively light weight, as in not impose a heavy infrastructure

- It should not be Microsoft excel – that should be obvious …

General calculation results

```
[ { 'type' : 'effectiveMultiplicationFactor’,
    'data' : { 'values' : [ 1.0000 ],
               'uncertainties' : [ 0.0001 ] } },
  { 'type' : 'sensitivityProfile',
    'response' : 'keff',
    'parameter' : 'xs',
    'particleId' : 'neutron',
    'nuclide' : '92235',
    'reaction' : 'fission',
    'material' : 'total',
    'data' : { 'values' : [ -1.7129e-17, 1.4106e-09 ],
               'uncertainties' : [ 0.0034, 0.0033 ],
               'structure' : [ { 'name' : 'energy-in',
                                  'type' : 'histogram',
                                  'limits' : [ 1e-11, 10.0, 20.0 ],
                                  'unit' : 'MeV'} ],
               'units' : { 'value' : '%/%', 'uncertainty' : 'relative' } } } ]
```

At LANL, we have started experimenting with only these two types of results but potential is limitless.

Python interface

```
# results is a list of Results equal to the list given in the previous slide

# get profile from the list
profile = results[1]

# retrieve attribute information with the 'attributes' property on Result
type = profile.attributes.type           # 'sensitivityProfile'
nuclide = profile.attributes.nuclide     # '92235'
reaction = profile.attributes.reaction # 'fission'
date = profile.attributes.date           # None, 'date' attribute is not present

# retrieve the values and group structure of the sensitivity profiles
values = profile.values                  # profile.data.values also works here
groups = profile.structure[0].limits
valueUnit = profile.units.value
```

Python interface

```
# filter on all results to get all sensitivity profiles for U235 xs
nuclide = '92235'
search = [ result for result in results
             if result.attributes.type == 'sensitivityProfile' and
                result.attributes.nuclide == nuclide ]

# retrieve and print all reactions of U235 for which we have profiles
reactions = [ result.attributes.reaction for result in search ]
print( reactions )   # [ 'fission' ]

# create new results from data
myKeffResult = makeKeffResult( 1.0001, 0.0005 )
mySensitivityProfile =
   makeKeffSensitivityResult( 'xs', '92235', 'fission',
                              Data( profile, uncertainties,
                                    dimensions, units ) )
```

Python interface

```
# we have an interface over MCNP and SENSMG outputs to use this structure
mcnp = McnpOutput( [ McnpEffectiveMultiplicationFactor(),
                     McnpSensitivityProfiles() ] )
mcnp.extract( 'HEU-MET-FAST-001-001.mcnp.o' )

results = {}
results[ 'HEU-MET-FAST-001-001' ] = mcnp.toResults()

# we have json serialisation and deserialization in place
toJSON( results, 'mcnp.results.json', indent = 2 )
resultsFromJSON = fromJSON( 'mcnp.results.json’ )

# results and resultsFromJSON are the same
print( results == resultsFromJSON ) # should be true
```

Attributes

`type` is an essential attribute and should be always be present

- Indicate the type of result we're storing
- Possible values that are currently identified: `effectiveMultiplicationFactor`, `effectiveDelayedNeutronFraction`, `particleSpectrum`, `particleFlux`, `particleCurrent`, `reactionRate`, `sensitivityProfile`

Some attributes will appear based on the value of `type`

- For example, for `sensitivityProfile` :
  - `response` : for which "response function" we have a sensitivity $\partial r/\partial p$, e.g. `keff`
  - `parameter` : the sensitivity of the response is given with respect to a parameter, e.g. `xs`, `spectrum` or `angular`
  - `nuclide` : the nuclide for which a sensitivity profile is given
  - `reaction` : the reaction for which the sensitivity profile is given, e.g. `fission`, `n,gamma`
  - `material` : the material in the model for which the sensitivity is given, e.g. `total` or a material identifier

Attributes

Some attributes could appear based on the application but should be independent of the value of `type`

For example (we do not use these yet):

- `code` : which calculation code generated the result, e.g. `mcnp`, `cog`, `partisan`, `ardra`
- `date` : the calculation date
- `library` : the nuclear data library, e.g. `endf/b-viii.0`
- `temperature` : the temperature of the material for which the result is given

The way attributes are stored and defined makes it flexible enough for extension

- Retrieving a non-existent attribute is NOT an error, it is simply undefined
- This allows for filters to function properly

Data

As indicated earlier, a calculation result consists of the following:

- A one dimensional array of `values`, this is always present
- An optional one dimensional array of `uncertainties`
- The `structure` of the `values` and `uncertainties`
  - This is a list of dimensions that defines how to interpret the the `values` and `uncertainties`
  - This is not required if the `values` array contains a single value
- An optional set of units, one for a `value` and another one for an `uncertainty`

`values` and `uncertainties` are always an array

- This even applies to a single value (every result needs to look like another one)

Dimensions and the structure of the result

The structure of a result is made up of dimensions, defined by:

- `name` : the name for the dimension, e.g. `energy-in`
- `type` : the type of the dimension, either `histogram` or `points`
- `limits` : the bins or points for which we have data in the current dimension
- `unit` : an optional unit for the dimension

A one dimensional result will have only one dimension, and so on

- A particle spectrum integrated over a given number energy bins has one dimension
- A sensitivity profile for the fission spectrum can have an incident energy dimension and an outgoing energy dimension
- The order of the dimensions determines the order of the values (obviously)

Dimensions and the structure of the result

The dimension `type` can be mixed over multiple dimensions

- I can use a first dimension that gives me points in time followed by a second dimension that gives me histograms for each point in time (e.g. changes in particle spectrum as a function of time)

```
[ { 'name' : 'time', 'type' : 'points',
    'limits' : [ 0, 1, 2 ], 'unit' : 'days' },
  { 'name' : 'energy', 'type' : 'histogram',
    'limits' : [ 1e-5, 1.0, 2e+7 ], 'unit' : 'eV' } ]
```

The number of values and uncertainties is directly linked to the dimension

- Dimensions must be present as soon as there is more than 1 value in the arrays