XSBoa
SERPENT Wrapper for Cross Section Processing

Andrew Johnson

27 Mar, 2017

1 Objective

The purpose of this document is to describe the proposed plan for creating a program that creates homogenized cross sections for use in nodal diffusion codes. This program, henceforth called XSBoa, will perform the following actions:

1. Read in a template SERPENT input file
2. Reproduce the input file with minor modifications, as to cover the design space of the problem including,
   - fuel temperature
   - burnup
   - moderator density (void)
   - control rod insertion
   - soluble absorber concentration
3. Execute each template file in a manner that takes advantage of a parallel environment
4. Scrape SERPENT result files to obtain homogenized cross sections and transport parameters (e.g. kinetic parameters, assembly discontinuity factors, diffusion, delayed neutron fraction, etc.)
5. Allow maximum control over what parameters the user desires
6. Present these values in format befitting multiple applications
   - Human-readable text
   - LaTeX-ready text
   - Direct input into nodal diffusion codes

2 Implementation

It is my preference to use Python 3.5 for this project, primarily due to Python's flexibility with string and file operations and the free nature of Python (vs. MATLAB) and secondarily due to my own comfort and skill with the language.

2.1 Initialization

I see three main operating regimes for XSBoa: **c**reation of input files, e**x**ecution of SERPENT, and **p**rocessing the output files

XSBoa **−file** [−c|−x|−p]

`[-c]` running `XSBoa` through the creation on modified `SERPENT` input files
`[-x]` creating and executing each modified input file
`[-p]` processing the outputs from completed `SERPENT` runs

where the optional flags `-c|-x|-p` correspond to I propose that the input parameter `file` is a valid, standalone `SERPENT` input file with header block, indicating parameters that control the branching calculations.

```
/* Start Boa block
        .
        .
        .
End Boa block */
```

So long as the beginning and ending of this data block can be identified simply, it can be read by the processing script, and then removed for each branching calculation.

2.2 Identify state points

For what I envision, `XSBoa` must be able to identify

1. Nominal operation conditions[1]
2. Modified conditions for branching
3. Burnup regimes
4. Universe swaps (i.e. control rod insertion or fuel assembly shuffling)

I propose the following syntax for the first three conditions:

```
nom <mat> <param> value
        nom fuel3 temp 900
        nom water1 dens -0.740
branch <bname> <mat> <param> <value>
        branch hFuel fuel3 temp 1200
        branch v90 water1 dens -0.074
burn <type> <schedule>
        burn istep 1 4 5 15 25 50R6
```

The nominal values would be the default parameters written to the input files. In the case where the branch modifies the nominal value, then the branch value is used instead. This could be accommodated in Python by using two dictionary containers for the nominal and branch states.

```
branchD = {'fuel3': ('temp', 1200)}
nomD = {'fuel3': ('temp', 900),
        'water1': ('dens', -0.740)}
if mat in branchD:
        # use values from branchD
elif mat in nomD:
        # use values from nomD
```

The burnup command would write appropriate `SERPENT` depletion cycles with the following keywords: [h!] In order to simplify the header file, one could either leave the depletion steps out of the header and write them directly into the `SERPENT` input file. Or, we could adopt syntax similar to `SCALE` and allow implicit repetition where the phrase `50R6` gets written as `50 50 50 50 50 50`.

[1] if the input file is constructed for the nominal case, then this option could be removed without any issue

istep	daystep	incremental days
cstep	daytot	cummulative days
iburn	bustep	incremental burnup interval
cburn	butot	cummulative burnup interval
idec	decstep	incremental decay time
cdec	dectot	cummulative decay time

2.3 Creation of Input Files

Up to this point, the design space has been specified through nominal and branching points. Now, for every individual branch calculation, a new input file will be written in the form

<input file>_<bname>

At this point, if the user executed XSBoa with the -c option, the code will successfully terminate. Otherwise, for each file created, XSBoa will execute the files according to

```
exe_str = 'qsub -N {0} -v inparg={0} sub_serpent.pbs'
for inp in files:
        subprocess.call[exe_str.format(inp)]
```

The .format function replaces every instance of {0} with the name of the file inp and calls the execution string from subprocess.call.

3 Post-Processing

Every branch file that is created will be added to a text file that lists the name of the file, status of the operation (0 for not processed, -1 for errors, 1 if data has been processed), and branch states. Using this file, XSBoa executed with the -p flag would be able to quickly find the corresponding _res.m files and scrape the desired data. This table indicates that the branch hFuel has not been processed yet, and the desired

Branch Name	Status	Text output
hFuel	0	None
v90	1	outputs/test_v90_t.txt

outputs from branch v90 are in the file test_v90_t.txt in directory outputs.

Included in the XSBoa header block would be parameters indicating which values to extract from the output files. These could be lumped into groups that would commonly be used together. Multigroup energy spectrum would be automatically extracted.

INF	Infinite spectrum group constants
B1	Leakage corrected group constants
KIN	Delayed neutron parameters and point kinetics values
ADF	Assembly discontinuity factors

Once a file has been processed, the corresponding status value in the pointer text file would be updated to 1 if successful, -1 for errors. These parameters would be written to a single, human-readable text file for verification. An additional input parameter -f could be added to direct XSBoa to write the values in a specific format, i.e. for use in nodal diffusion codes. This section would require reading the output files in as class objects, containers for desired parameters, and methods for reading and writing the data.