serpentTools Documentation

Release 0.2.1

Andrew Johnson, Dan Kotlyar, Stefano Terlizzi

January 12, 2018

CONTENTS:

WELCOME!

serpent-tools

A suite of parsers designed to make interacting with `SERPENT` [1] output files simple and flawless.

The `SERPENT` Monte Carlo code is developed by VTT Technical Research Centre of Finland, Ltd. More information, including distribution and licensing of `SERPENT` can be found at montecarlo.vtt.fi

Installation

The `serpentTools` package can be downloaded either as a git repository or as a zipped file. Both can be obtained through the `Clone or download` option at the serpent-tools GitHub.

Once the repository has been downloaded or extracted from zip, the package can be installed with:

```
cd serpentTools
python setup.py install
python setup.py test
```

Installing with setuptools is preferred over the standard `distutils` module. `setuptools` can be installed with `pip` as:

```
pip install -U setuptools
```

Installing in this manner ensures that the supporting packages, like `numpy` are installed and up to date.

Issues

If you have issues installing the project, find a bug, or want to add a feature, the GitHub issue page is the best place to do that.

Contributors

Here are all the wonderful people that helped make this project happen

- Andrew Johnson
- Dr. Dan Kotlyar
- Stefano Terlizzi

---

[1] Leppanen, J. et al. (2015) "The Serpent Monte Carlo code: Status, development and applications in 2013." Ann. Nucl. Energy, 82 (2015) 142-150

References

The Annals of Nuclear Energy article should be cited for all work using `SERPENT`. If you wish to cite this project, please cite as:

```
url{@serpentTools
    author = {Andrew Johnson and Dan Kotlyar and Stefano Terlizzi},
    title = {serpentTools: A suite of parsers designed to make
    interacting with SERPENT outputs simple and flawless},
    url = {https://github.com/CORE-GATECH-GROUP/serpent-tools},
    year = {2017}
}
```

License

MIT License

Copyright (c) 2017-2018 Andrew Johnson, Dan Kotlyar, Stefano Terlizzi, GTRC

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

EXAMPLES

User Control

The `serpentTools` package is designed to, without intervention, be able to store all the data contained in each of the various output files. However, the `serpentTools.settings` module grants great flexibility to the user over what data is obtained through the rc class. This notebook will provide as an intro into using this class.

Basic Usage

```
>>> import serpentTools
INFO   : serpentTools: Using version 0.2.1
>>> from serpentTools.settings import rc, defaultSettings
```

Below are the default values for each setting available

```
>>> for setting in sorted(defaultSettings.keys()):
...     print(setting)
...     for key in defaultSettings[setting]:
...         print('\t', key, '-', defaultSettings[setting][key])
depletion.materialVariables
    default - []
    description - Names of variables to store. Empty list -> all variables.
    type - <class 'list'>
depletion.materials
    default - []
    description - Names of materials to store. Empty list -> all materials.
    type - <class 'list'>
depletion.metadataKeys
    default - ['ZAI', 'NAMES', 'DAYS', 'BU']
    description - Non-material data to store, i.e. zai, isotope names, burnup schedule, etc.
    type - <class 'list'>
    options - default
depletion.processTotal
    default - True
    description - Option to store the depletion data from the TOT block
    type - <class 'bool'>
serpentVersion
    default - 2.1.29
    description - Version of SERPENT
    type - <class 'str'>
    options - ['2.1.29']
verbosity
    default - warning
```

```
     type – <class 'str'>
     description – Set the level of errors to be shown.
     updater – <function updateLevel at 0x000001B7F3DD6598>
     options – ['critical', 'error', 'warning', 'info', 'debug']
xs.variableExtras
     default – []
     description – Full SERPENT name of variables to be read
     type – <class 'list'>
xs.variableGroups
     default – []
     description – Name of variable groups from variables.yaml to be expanded into SERPENT variable t
     type – <class 'list'>
```

Settings such as `depletion.materialVariables` are specific for the `DepletionReader`, while settings that are led with `xs` are sent to the `ResultsReader` and `BranchingReader`, as well as their specific settings. The `rc` class acts as a dictionary, and updating a value is as simple as

```
>> rc['verbosity'] = 'debug'
DEBUG   : serpentTools: Updated setting verbosity to debug
```

The `rc` object automatically checks to make sure the value is of the correct type, and is an allowable option, if given.

```
>>> try:
...     rc['depletion.metadataKeys'] = False
... except TypeError as te:
...     print(te)
Setting depletion.metadataKeys should be of type <class 'list'>, not <class 'bool'>
>>> try:
...     rc['serpentVersion'] = '1.2.3'
... except KeyError as ke:
...     print(ke)
"Setting serpentVersion is
1.2.3
and not one of the allowed options:
['2.1.29']"
```

The `rc` object can also be used inside a context manager to revert changes.

```
>>> with rc:
...     rc['depletion.metadataKeys'] = ['ZAI', 'BU']
>>>
>>> rc['depletion.metadataKeys']
>>> rc['verbosity'] = 'info'
DEBUG   : serpentTools: Updated setting depletion.metadataKeys to ['ZAI', 'BU']
DEBUG   : serpentTools: Updated setting depletion.metadataKeys to ['ZAI', 'NAMES', 'DAYS', 'BU']
['ZAI', 'NAMES', 'DAYS', 'BU']
```

Group Constant Variables

Two settings control what group constant data and what variables are extracted from the results and coefficient files.

1. `xs.variableExtras`:   Full   `SERPENT_STYLE`   variable   names,   i.e.   `INF_TOT`, `FISSION_PRODUCT_DECAY_HEAT`

2. `xs.variableGroups`: Select keywords that represent blocks of common variables

These variable groups are stored in `serpentTools/variables.yaml` and rely upon the `SERPENT` version to properly expand the groups.

```
>>> rc['serpentVersion']
'2.1.29'
>>> rc['xs.variableGroups'] = ['kinetics', 'xs', 'diffusion']
>>> rc['xs.variableExtras'] = ['XS_DATA_FILE_PATH']
>>> varSet = rc.expandVariables()
>>> print(sorted(varSet))
['ABS', 'ADJ_IFP_ANA_BETA_EFF', 'ADJ_IFP_ANA_LAMBDA', 'ADJ_IFP_GEN_TIME',
 'ADJ_IFP_IMP_BETA_EFF', 'ADJ_IFP_IMP_LAMBDA', 'ADJ_IFP_LIFETIME',
 'ADJ_IFP_ROSSI_ALPHA', 'ADJ_INV_SPD', 'ADJ_MEULEKAMP_BETA_EFF',
 'ADJ_MEULEKAMP_LAMBDA', 'ADJ_NAUCHI_BETA_EFF', 'ADJ_NAUCHI_GEN_TIME',
 'ADJ_NAUCHI_LAMBDA', 'ADJ_NAUCHI_LIFETIME', 'ADJ_PERT_BETA_EFF',
 'ADJ_PERT_GEN_TIME', 'ADJ_PERT_LIFETIME', 'ADJ_PERT_ROSSI_ALPHA', 'CAPT',
 'CHID', 'CHIP', 'CHIT', 'CMM_DIFFCOEF', 'CMM_DIFFCOEF_X', 'CMM_DIFFCOEF_Y',
 'CMM_DIFFCOEF_Z', 'CMM_TRANSPXS', 'CMM_TRANSPXS_X', 'CMM_TRANSPXS_Y',
 'CMM_TRANSPXS_Z', 'DIFFCOEF', 'FISS', 'FWD_ANA_BETA_ZERO',
 'FWD_ANA_LAMBDA', 'INVV', 'KAPPA', 'NSF', 'NUBAR', 'RABSXS', 'REMXS',
 'S0', 'S1', 'S2', 'S3', 'S4', 'S5', 'S6', 'S7', 'SCATT0', 'SCATT1',
 'SCATT2', 'SCATT3', 'SCATT4', 'SCATT5', 'SCATT6', 'SCATT7', 'TOT',
 'TRANSPXS', 'XS_DATA_FILE_PATH']
```

However, one might see that the full group constant cross sections are not present in this set

```
>>> assert 'INF_SCATT3' not in varSet
```

This is because two additional settings instruct the *BranchingReader* and *ResultsReader* to obtain infinite medium and leakage-corrected cross sections: `xs.getInfXS` and `xs.getB1XS`, respectively. By default, `xs.getInfXS` and `xs.getB1XS` default to True. This, in conjunction with leaving the `xs.variableGroups` and `xs.variableExtras` settings to empty lists, instructs these readers to obtain all the data present in their respective files.

See the *Branching Reader* example for more information on using these settings to control scraped data.

Configuration Files

As of version 0.1.2, the `rc` object allows for settings to be updated from a yaml configuration file using the *loadYaml()* method. The file is structured with the names of settings as keys and the desired setting value as the values. The loader also attempts to expand nested settings, like reader-specific settings, that may be lumped in a second level.

```
verbosity: warning
xs.getInfXS: False
branching:
    areUncsPresent: False
    floatVariables: [Fhi, Blo]
depletion:
    materials: [fuel*]
    materialVariables:
        [ADENS, MDENS, VOLUME]
```

Branching Reader

This notebook demonstrates the capability of the serpentTools package to read branching coefficient files. The format of these files is structured to iterate over:

1. Branch states, e.g. burnup, material properties

2. Homogenized universes

3. Group constant data

The output files are described in more detail on the SERPENT Wiki

Basic Operation

The simplest way to read these files is using the `serpentTools.parsers.read()` function

---

Note: Without modifying the settings, the `BranchingReader` assumes that all group constant data is presented without the associated uncertainties. See *User Control* for examples on the various ways to control operation

---

```
>>> import serpentTools
>>> branchFile = 'demo.coe'
INFO    : serpentTools: Using version 0.2.1
>>> r0 = serpentTools.read(branchFile)
INFO    : serpentTools: Inferred reader for demo.coe: BranchingReader
INFO    : serpentTools: Preparing to read demo.coe
INFO    : serpentTools: Done reading branching file
```

The branches are stored in custom `BranchContainer` objects in the `branches` dictionary

```
>>> r0.branches
{('B1000', 'FT1200'):
    <serpentTools.objects.containers.BranchContainer at 0x2220c762438>,
 ('B1000', 'FT600'):
    <serpentTools.objects.containers.BranchContainer at 0x2220c787908>,
 ('B1000', 'nom'):
    <serpentTools.objects.containers.BranchContainer at 0x2220c737ef0>,
 ('B750', 'FT1200'):
    <serpentTools.objects.containers.BranchContainer at 0x2220c752cf8>,
 ('B750', 'FT600'):
    <serpentTools.objects.containers.BranchContainer at 0x2220c77c208>,
 ('B750', 'nom'):
    <serpentTools.objects.containers.BranchContainer at 0x2220c72c860>,
 ('nom', 'FT1200'):
    <serpentTools.objects.containers.BranchContainer at 0x2220c7455f8>,
 ('nom', 'FT600'):
    <serpentTools.objects.containers.BranchContainer at 0x2220c76dac8>,
 ('nom', 'nom'):
    <serpentTools.objects.containers.BranchContainer at 0x2220c7231d0>}
```

Here, the keys are tuples of strings indicating what perturbations/branch states were applied for each SERPENT solution. Examining a particular case

```
>>> b0 = r0.branches['B1000', 'FT600']
>>> print(b0)
<BranchContainer for B1000, FT600 from demo.coe>
```

SERPENT allows the user to define variables for each branch through:

```
var V1_name V1_value
```

cards. These are stored in the `stateData` attribute

```
>>> b0.stateData
{'BOR': '1000',
 'DATE': '17/12/19',
 'TFU': '600',
 'TIME': '09:48:54',
 'VERSION': '2.1.29'}
```

The keys 'DATE', 'TIME', and 'VERSION' are included by default in the output, while the 'BOR' and 'TFU' have been defined for this branch.

Group Constant Data

Note: Group constants are converted from SERPENT_STYLE to mixedCase to fit the overall style of the project.

The *BranchContainer* stores group constant data in *HomogUniv* objects in the universes dictionary

```
>>> b0.universes
{(0, 0.0, 1): <serpentTools.objects.containers.HomogUniv at 0x2220c781ac8>,
 (0, 1.0, 2): <serpentTools.objects.containers.HomogUniv at 0x2220c78b5f8>,
 (0, 10.0, 3): <serpentTools.objects.containers.HomogUniv at 0x2220c791240>,
 (10, 0.0, 1): <serpentTools.objects.containers.HomogUniv at 0x2220c787a58>,
 (10, 1.0, 2): <serpentTools.objects.containers.HomogUniv at 0x2220c78b6a0>,
 (10, 10.0, 3): <serpentTools.objects.containers.HomogUniv at 0x2220c791320>,
 (20, 0.0, 1): <serpentTools.objects.containers.HomogUniv at 0x2220c787cc0>,
 (20, 1.0, 2): <serpentTools.objects.containers.HomogUniv at 0x2220c78b908>,
 (20, 10.0, 3): <serpentTools.objects.containers.HomogUniv at 0x2220c791588>,
 (30, 0.0, 1): <serpentTools.objects.containers.HomogUniv at 0x2220c78b048>,
 (30, 1.0, 2): <serpentTools.objects.containers.HomogUniv at 0x2220c78bb70>,
 (30, 10.0, 3): <serpentTools.objects.containers.HomogUniv at 0x2220c7917f0>,
 (40, 0.0, 1): <serpentTools.objects.containers.HomogUniv at 0x2220c78b1d0>,
 (40, 1.0, 2): <serpentTools.objects.containers.HomogUniv at 0x2220c78bdd8>,
 (40, 10.0, 3): <serpentTools.objects.containers.HomogUniv at 0x2220c791a58>}
```

The keys here are vectors indicating the universe ID, burnup [MWd/kgU], and burnup index corresponding to the point in the burnup schedule. These universes can be obtained by indexing this dictionary, or by using the *getUniv()* method

```
>>> univ0 = b0.universes[0, 1, 2]
>>> print(univ0)
>>> print(univ0.name)
>>> print(univ0.bu)
>>> print(univ0.step)
>>> print(univ0.day)
<HomogUniv from demo.coe>
0
1.0
2
0
>>> univ1 = b0.getUniv(0, burnup=1)
>>> univ2 = b0.getUniv(0, index=2)
>>> assert univ0 is univ1 is univ2
```

Since the coefficient files do not store the day value of burnup, all *HomogUniv* objects created by the *BranchContainer* default to day zero.

Group constant data is stored in five dictionaries:

---

1. `infExp`: Expected values for infinite medium group constants

2. `infUnc`: Relative uncertainties for infinite medium group constants

3. `b1Exp`: Expected values for leakge-corrected group constants

4. `b1Unc`: Relative uncertainties for leakge-corrected group constants

5. `metaData`: items that do not fit the in the above categories

```
>>> univ0.infExp
{'infDiffcoef': array([ 1.83961 ,   0.682022]),
 'infFiss': array([ 0.00271604,  0.059773  ]),
 'infRem': array([], dtype=float64),
 'infS0': array([ 0.298689  ,   0.00197521,  0.00284247,  0.470054  ]),
 'infS1': array([ 0.0847372 ,   0.00047366,  0.00062865,  0.106232  ]),
 'infTot': array([ 0.310842,  0.618286])}
>>> univ0.infUnc
{}
>>> univ0.b1Exp
{'b1Diffcoef': array([ 1.79892 ,   0.765665]),
 'b1Fiss': array([ 0.00278366,  0.0597712 ]),
 'b1Rem': array([], dtype=float64),
 'b1S0': array([ 0.301766  ,   0.0021261 ,  0.00283866,  0.470114  ]),
 'b1S1': array([ 0.0856397 ,   0.00051071,  0.00062781,  0.106232  ]),
 'b1Tot': array([ 0.314521,  0.618361])}
>>> univ0.metaData
{}
```

Group constants and their associated uncertainties can be obtained using the `get()` method.

```
>>> univ0.get('infFiss')
array([ 0.00286484,  0.0577559 ])
>>> try:
>>>     univ0.get('infS0', uncertainty=True)
>>> except KeyError as ke:  # no uncertainties here
>>>     print(str(ke))
'Variable infS0 absent from uncertainty dictionary'
```

Iteration

The branching reader has a `iterBranches()` method that works to yield branch names and their associated `BranchContainer` objects. This can be used to efficiently iterate over all the branches presented in the file.

```
>>> for names, branch in r0.iterBranches():
>>>     print(names, branch)
('nom', 'nom') <BranchContainer for nom, nom from demo.coe>
('B750', 'nom') <BranchContainer for B750, nom from demo.coe>
('B1000', 'nom') <BranchContainer for B1000, nom from demo.coe>
('nom', 'FT1200') <BranchContainer for nom, FT1200 from demo.coe>
('B750', 'FT1200') <BranchContainer for B750, FT1200 from demo.coe>
('B1000', 'FT1200') <BranchContainer for B1000, FT1200 from demo.coe>
('nom', 'FT600') <BranchContainer for nom, FT600 from demo.coe>
('B750', 'FT600') <BranchContainer for B750, FT600 from demo.coe>
('B1000', 'FT600') <BranchContainer for B1000, FT600 from demo.coe>
```

User Control

The SERPENT set coefpara card already restricts the data present in the coeffient file to user control, and the *BranchingReader* includes similar control. Below are the various settings that the *BranchingReader* uses to read and process coefficient files.

```
>>> import six
>>> from serpentTools.settings import rc
>>> from serpentTools.settings import rc, defaultSettings
>>> for setting in defaultSettings:
>>>     if 'xs' in setting or 'branching' in setting:
>>>         print(setting)
>>>         for k, v in six.iteritems(defaultSettings[setting]):
>>>             print('\t', k+':', v)
branching.areUncsPresent
    default: False
    type: <class 'bool'>
    description: True if the values in the .coe file contain uncertainties
branching.intVariables
    default: []
    description: Name of state data variables to convert to integers for
    each branch
    type: <class 'list'>
branching.floatVariables
    default: []
    description: Names of state data variables to convert to floats for
    each branch
    type: <class 'list'>
xs.getInfXS
    default: True
    description: If true, store the infinite medium cross sections.
    type: <class 'bool'>
xs.getB1XS
    default: True
    description: If true, store the critical leakage cross sections.
    type: <class 'bool'>
xs.variableGroups
    default: []
    description: Name of variable groups from variables.yaml to be expanded
     into SERPENT variable to be stored
    type: <class 'list'>
xs.variableExtras
    default: []
    description: Full SERPENT name of variables to be read
    type: <class 'list'>
```

In our example above, the BOR and TFU variables represented boron concentration and fuel temperature, and can easily be cast into numeric values using the branching.intVariables and brancing.floatVariables settings. From the previous example, we see that the default action is to store all state data variables as strings.

```
>>> assert isinstance(b0.stateData['BOR'], str)
```

As demonstrated in the *Group Constant Variables* example, use of xs.variableGroups and xs.variableExtras controls what data is stored on the *HomogUniv* objects. By default, all variables present in the coefficient file are stored.

```
>>> rc['branching.floatVariables'] = ['BOR']
>>> rc['branching.intVariables'] = ['TFU']
>>> rc['xs.getB1XS'] = False
```

```
>>> rc['xs.variableExtras'] = ['INF_TOT', 'INF_SCATT0']
>>> r1 = serpentTools.read(branchFile)
INFO    : serpentTools: Inferred reader for demo.coe: BranchingReader
INFO    : serpentTools: Preparing to read demo.coe
INFO    : serpentTools: Done reading branching file
>>> b1 = r1.branches['B1000', 'FT600']
>>> b1.stateData
{'BOR': 1000.0,
 'DATE': '17/10/18',
 'TFU': 600,
 'TIME': '10:26:48',
 'VERSION': '2.1.29'}
>>> assert isinstance(b1.stateData['BOR'], float)
>>> assert isinstance(b1.stateData['TFU'], int)
```

Inspecting the data stored on the homogenized universes reveals only the variables explicitly requested are present

```
>>> univ4 = b1.getUniv(0, 0)
>>> univ4.infExp
{'infTot': array([ 0.313338,   0.54515 ])}
>>> univ4.b1Exp
{}
```

Conclusion

The *BranchingReader* is capable of reading coefficient files created by the SERPENT automated branching process. The data is stored according to the branch parameters, universe information, and burnup. This reader also supports user control of the processing by selecting what state parameters should be converted from strings to numeric types, and further down-selection of data.

DetectorReader

Basic Operation

The *DetectorReader* is capable of reading SERPENT detector files. These detectors can be defined with many binning parameters, listed on the SERPENT Wiki. One could define a detector that has a spatial mesh, $dx/dy/dz/$, but also includes reaction and material bins, dr, dm. Detectors are stored on the reader object in the detectors dictionary as custom *Detector* objects. Here, all energy and spatial grid data are stored, including other binning information such as reaction, universe, and lattice bins.

```
>>> import serpentTools
INFO    : serpentTools: Using version 0.2.1
>>> detFile = 'demo_det0.m'
>>> det = serpentTools.read(detFile)
INFO    : serpentTools: Inferred reader for demo_det0.m: DetectorReader
INFO    : serpentTools: Preparing to read demo_det0.m
INFO    : serpentTools: Done
>>> det.detectors
{'nodeFlx': <serpentTools.objects.containers.Detector at 0x19e3c4f6b70>,
 'spectrum': <serpentTools.objects.containers.Detector at 0x19e3c4f6ba8>,
 'xyFissionCapt':
    <serpentTools.objects.containers.Detector at 0x19e3c4f6470>}
```

These detectors were defined to a single fuel pin, divided into 16 axial segments by unique materials. A description of the detectors provided in the output:

Name	Description
nodeFlx	Flux-share in each axial universe
spectrum	1968 group flux in fuel pin
xyFissionCapt	One-group fission and capture rate for a 5x5 xy grid

For each `Detector` object, the full tally matrix is stored in the `bins` array.

```
>>> nodeFlx = det.detectors['nodeFlx']
>>> print(nodeFlx.bins.shape)
(16, 12)
>>> nodeFlx.bins[:5,:]
array([[  1.00000000e+00,   1.00000000e+00,   1.00000000e+00,
          1.00000000e+00,   1.00000000e+00,   1.00000000e+00,
          1.00000000e+00,   1.00000000e+00,   1.00000000e+00,
          1.00000000e+00,   2.34759000e-02,   4.53000000e-03],
       [  2.00000000e+00,   1.00000000e+00,   2.00000000e+00,
          1.00000000e+00,   1.00000000e+00,   1.00000000e+00,
          1.00000000e+00,   1.00000000e+00,   1.00000000e+00,
          1.00000000e+00,   5.75300000e-02,   3.38000000e-03],
       [  3.00000000e+00,   1.00000000e+00,   3.00000000e+00,
          1.00000000e+00,   1.00000000e+00,   1.00000000e+00,
          1.00000000e+00,   1.00000000e+00,   1.00000000e+00,
          1.00000000e+00,   8.47000000e-02,   2.95000000e-03],
       [  4.00000000e+00,   1.00000000e+00,   4.00000000e+00,
          1.00000000e+00,   1.00000000e+00,   1.00000000e+00,
          1.00000000e+00,   1.00000000e+00,   1.00000000e+00,
          1.00000000e+00,   1.02034000e-01,   2.63000000e-03],
       [  5.00000000e+00,   1.00000000e+00,   5.00000000e+00,
          1.00000000e+00,   1.00000000e+00,   1.00000000e+00,
          1.00000000e+00,   1.00000000e+00,   1.00000000e+00,
          1.00000000e+00,   1.10384000e-01,   2.31000000e-03]])
```

Here, only three columns are changing:

- column 0: universe column

- column 10: tally column

- column 11: errors

Note: For SERPENT-1, there would be an additional column 12 that contained the scores for each bin

---

```
>>> nodeFlx.bins[:, 0]
array([  1.,    2.,    3.,    4.,    5.,    6.,    7.,    8.,    9.,   10.,   11.,
        12.,   13.,   14.,   15.,   16.])
```

Once each detector is given this binned tally data, the `reshape()` method is called to recast the `tallies`, `errors`, and, if applicable, the `scores` columns into individual, multidimensional arrays. For this case, since the only variable bin quantity is that of the universe, these will all be 1D arrays.

```
>>> assert nodeFlx.tallies.shape == (16, )
>>> assert nodeFlx.errors.shape == (16, )
>>> nodeFlx.tallies
array([ 0.0234759 ,   0.05753  ,   0.0847   ,   0.102034 ,   0.110384  ,
        0.110174  ,   0.102934 ,   0.0928861,   0.0810541,   0.067961  ,
        0.0550446 ,   0.0422486,   0.0310226,   0.0211475,   0.0125272 ,
```

```
        0.00487726])
>>> nodeFlx.errors
array([ 0.00453,  0.00338,  0.00295,  0.00263,  0.00231,  0.00222,
        0.00238,  0.00251,  0.00282,  0.00307,  0.00359,  0.00415,
        0.00511,  0.00687,  0.00809,  0.01002])
```

Bin information is retained through the `indexes` attribute. This is an *OrderedDictionary*, as the keys are placed according to their column position. These postions can be found in the SERPENT Manual, and are provided in the `DET_COLS` tuple

```
>>> from serpentTools.objects.containers import DET_COLS
>>> print(DET_COLS)
('value', 'energy', 'universe', 'cell', 'material', 'lattice',
 'reaction', 'zmesh', 'ymesh', 'xmesh', 'tally', 'error', 'scores')
>>> nodeFlx.indexes
OrderedDict([('universe',
            array([  1.,    2.,    3.,    4.,    5.,    6.,    7.,    8.,
                     9.,   10.,   11.,   12.,   13.,   14.,   15.,   16.]))])
```

Each item in the `indexes` ordered dictionary corresponds to the unique values of that bin in the original `bins` array. Here, `universe` is the first item and contains an equal number of elements to the size of the first (and only) axis in the nodeFlx tally matrix

```
>>> assert nodeFlx.indexes['universe'].size == nodeFlx.tallies.size
```

Plotting Routines

Each *Detector* object is capable of simple 1D and 2D plotting routines. The simplest 1D plot method is simply *plot()*, however a wide range of plot options are supported.
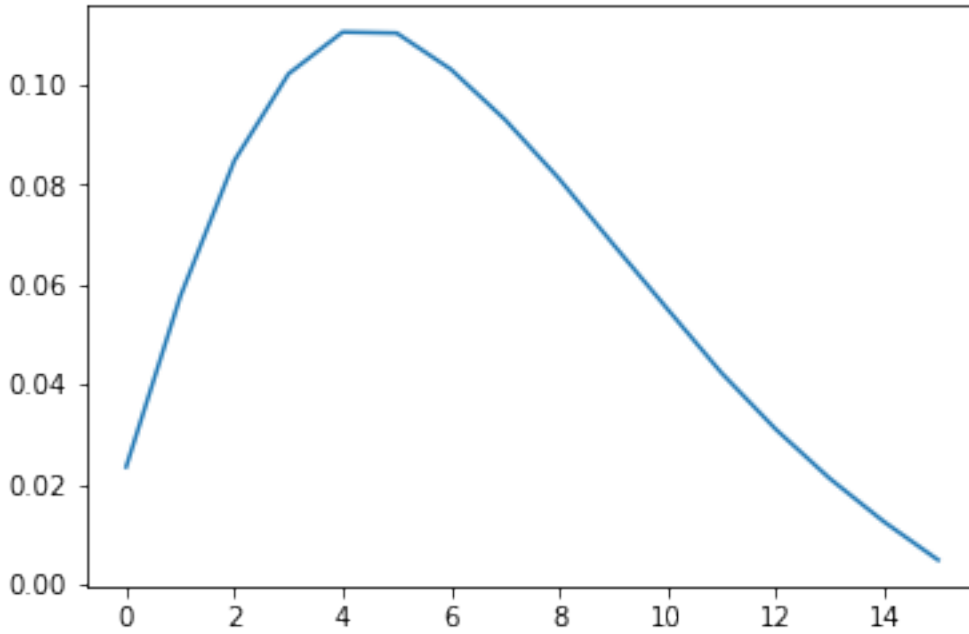
Option	Description
what	What data to plot
ax	Pre-prepared figure on which to add this plot
xdim	Quantity from indexes to use as x-axis
sigma	Confidence interval to place on errors
steps	Draw tally values as constant inside bin

The plot routine also accepts various options, which can be found in the matplotlib.pyplot.plot documentation
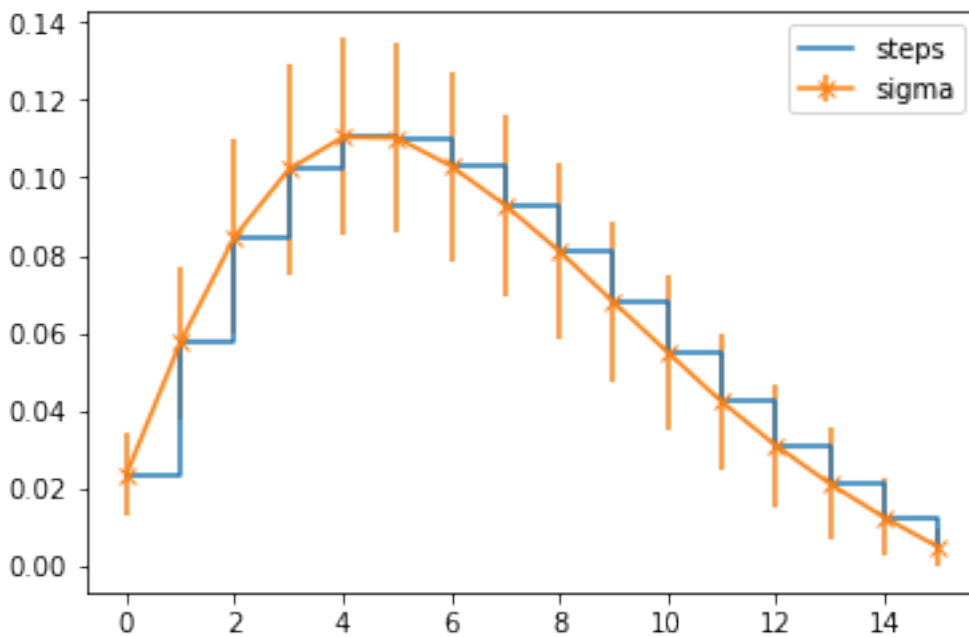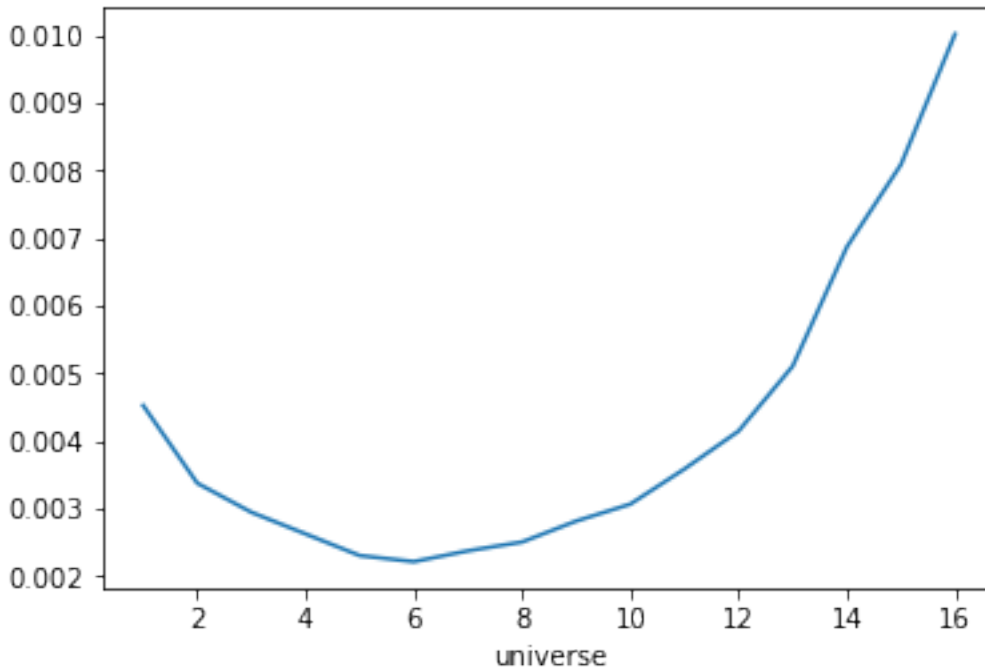
```
>>> nodeFlx.plot()
```

```
>>> ax = nodeFlx.plot(steps=True, label='steps')
>>> nodeFlx.plot(sigma=100, ax=ax, marker='x', label='sigma')
>>> ax.legend()
```


Passing `what='errors'` to the plot method plots the associated relative errors, rather than the tally data on the y-axis. Similarly, passing a key from `indexes` sets the x-axis to be that specific index.

```
>>> nodeFlx.plot(xdim='universe', what='errors');
```


For detectors that include some grid matrices, such as spatial or energy meshes DET<name>E, these arrays are stored in the grids dictionary

```
>>> spectrum = det.detectors['spectrum']
>>> print(spectrum.grids['E'][:5, :])
[[  1.00001000e-11   3.00000000e-09   1.50500000e-09]
 [  3.00000000e-09   5.00000000e-09   4.00000000e-09]
 [  5.00000000e-09   6.90000000e-09   5.95000000e-09]
 [  6.90000000e-09   1.00000000e-08   8.45000000e-09]
 [  1.00000000e-08   1.50000000e-08   1.25000000e-08]]
```

The *spectrumPlot()* method is designed to prepare plots of energy spectra. Supported keyword arguments method include

Option	Default	Description
normalize	True	Normalize tallies per unit lethargy
sigma	3	Level of confidence for statistical errors
xscale	'log'	Set the x scale to be log or linear
yscale	'linear'	Set the y scale to be log or linear

```
>>> spectrum.spectrumPlot()
```
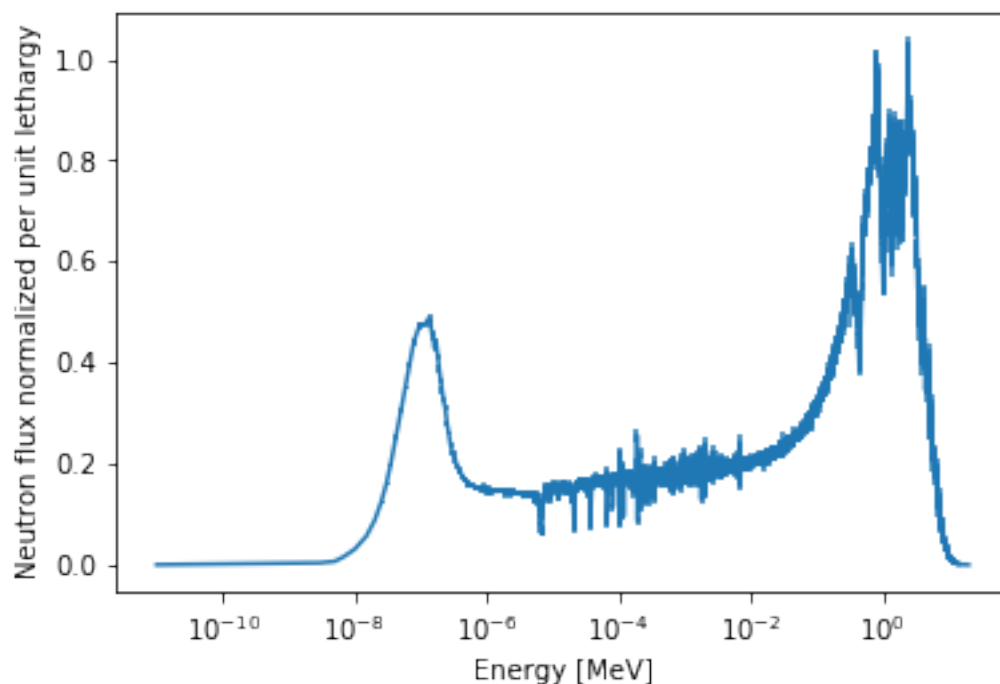
Multi-dimensional Detectors

The *Detector* objects are capable of reshaping the detector data intoan array where each axis corresponds to a varying bin. In the above examples, the reshaped data was one-dimensional, because the detectors only tallied data against one bin, universe and energy. In the following example, the detector has been configured to tally the fission and capture rates (two `dr` arguments) in an XY mesh.

```
>>> xy = det.detectors['xyFissionCapt']
>>> for key in xy.indexes:
...     print(key, xy.indexes[key])
reaction [ 1.  2.]
ymesh [ 1.  2.  3.  4.  5.]
xmesh [ 1.  2.  3.  4.  5.]
```

Traversing the first axis in the `tallies` array corresponds to changing the value of the `reaction`. The second axis corresponds to changing `ymesh` values, and the final axis reflects changes in `xmesh`.

```
>>> print(xy.bins.shape)
(50, 12)
>>> print(xy.tallies.shape)
(2, 5, 5)
>>> print(xy.bins[:5, 10])
[ 0.255119  0.255077  0.253685  0.255592  0.25845 ]
>>> print(xy.tallies[0, 0, :5])
[ 0.255119  0.255077  0.253685  0.255592  0.25845 ]
```

Slicing

As the detectors produced by SERPENT can contain multiple bin types, as seen in `DET_COLS`, obtaining data from the tally data can become complicated. This retrieval can be simplified using the *slice()* method. This method takes an argument indicating what bins (keys in `indexes`) to fix at what position.

If we want to retrieve the tally data for the XY mesh for the fission reactions

```
>>> xy.slice({'reaction': 1})
array([[ 0.255119,  0.255077,  0.253685,  0.255592,  0.25845 ],
       [ 0.254101,  0.253408,  0.256666,  0.255375,  0.252936],
       [ 0.256006,  0.251002,  0.255479,  0.252002,  0.254708],
       [ 0.254957,  0.253399,  0.24818 ,  0.252915,  0.253914],
       [ 0.258394,  0.250217,  0.259642,  0.254025,  0.257076]])
```

As the fission reaction corresponded to reaction tally 1 in the original matrix.

This method also works for slicing the error, or score, matrix

```
>>> xy.slice({'reaction': 2, 'ymesh': 4}, 'errors')
array([ 0.01576,  0.01299,  0.01811,  0.01367,  0.01632])
```

The method returned the relative error in the capture rate, along the fourth y-mesh,index 3 in the grid, for all x mesh points.
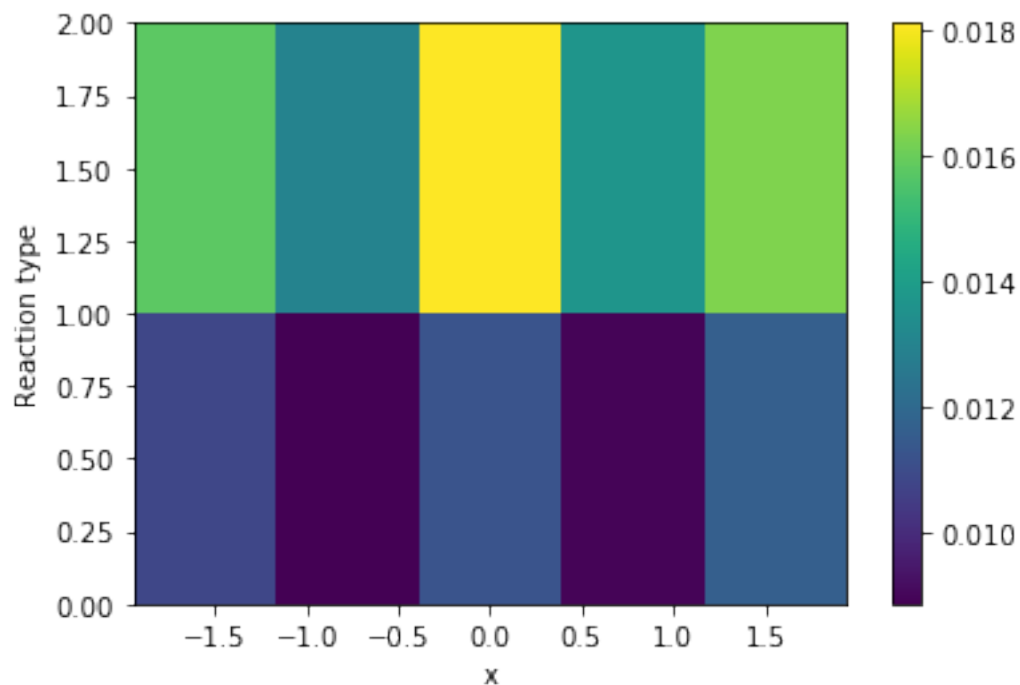
Plotting

For data with dimensionality greater than one, the *meshPlot()* method can be used to plot some 2D slice of the data.

```
>>> xy.meshPlot('x', 'y',fixed={'reaction':1})
```
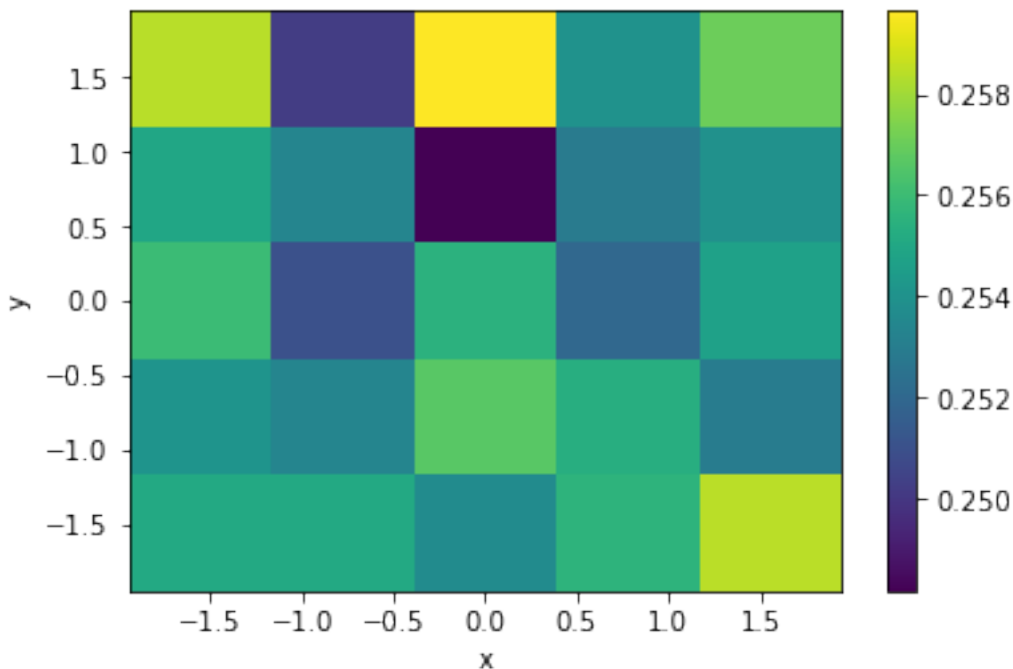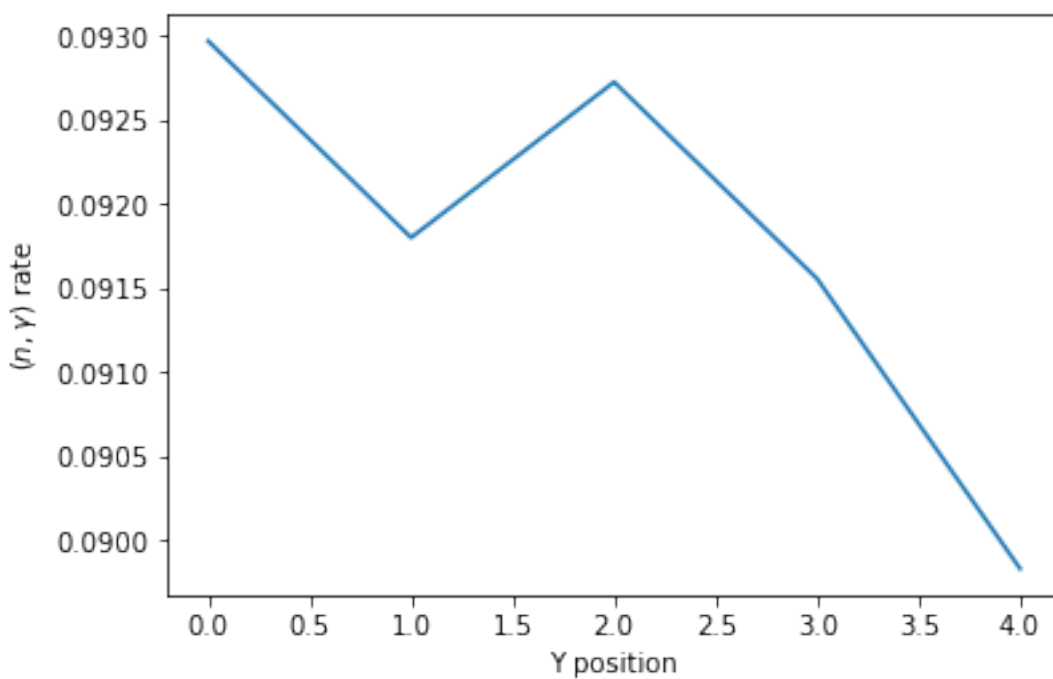

The *meshPlot()* also supports a range of labeling options

```
>>> xy.meshPlot('x', 'reaction', what='errors',
...             fixed={'ymesh': 4}, ylabel='Reaction type')
```

Using the `fixed` argument allows access to the 1D plot methods from before

```
>>> xy.plot(fixed={'reaction': 2, 'xmesh': 2},
...         xlabel='Y position',
...         ylabel=r'$\left(n,\gamma\right)$ rate');
```

Conclusion

The `DetectorReader` is capable of reading and storing detector data from SERPENT detector files. The data is stored on custom `Detector` objects, capable of reshaping tally and error matrices into arrays with dimensionality reflecting the detector binning. These `Detector` objects have simple methods for retrieving and plotting detector data.

References

1. J. Leppänen, M. Pusa, T. Viitanen, V. Valtavirta, and T. Kaltiaisenaho. "The Serpent Monte Carlo code: Status, development and applications in 2013." Ann. Nucl. Energy, 82 (2015) 142-150

DepletionReader

Basic Operation

SERPENT produces a burned material file, containing the evolution of material properties through burnup for all burned materials present in the problem. The `DepletionReader` is capable of reading this file, and storing the data inside `DepletedMaterial` objects. Each such object has methods and attributes that should ease analysis.

```
>>> import six
>>> import serpentTools
>>> from serpentTools.settings import rc
INFO    : serpentTools: Using version 0.2.1
>>> depFile = 'demo_dep.m'
>>> dep = serpentTools.read(depFile)
INFO    : serpentTools: Inferred reader for demo_dep.m: DepletionReader
INFO    : serpentTools: Preparing to read demo_dep.m
INFO    : serpentTools: Done reading depletion file
```

The materials read in from the file are stored in the `materials` dictionary, where the keys represent the name of specific materials, and the corresponding values are the depleted material.

```
>>> dep.materials
{'bglass0': <serpentTools.objects.materials.DepletedMaterial at 0x23905154668>,
 'fuel0': <serpentTools.objects.materials.DepletedMaterial at 0x2390578eeb8>,
 'total': <serpentTools.objects.materials.DepletedMaterial at 0x2390579e978>}
```

Metadata, such as the isotopic vector and depletion schedule are also present inside the reader

```
>>> dep.metadata.keys()
dict_keys(['zai', 'burnup', 'names', 'days'])
>>> dep.metadata['burnup']
array([ 0.  ,  0.02,  0.04,  ...,  1.36,  1.38,  1.4 ,  1.42])
>>> dep.metadata['names']
['Xe135', 'I135', 'U234', 'U235', 'U236', 'U238', 'Pu238',
 'Pu239',..., 'lost', 'total']
```

DepletedMaterial

As mentioned before, all the material data is stored inside these `DepletedMaterial` objects. These objects share access to the metadata of the reader as well.

```
>>> fuel = dep.materials['fuel0']
>>> fuel.burnup
array([ 0.  ,  0.02,  0.04,  ...,  1.36,  1.38,  1.4 ,  1.42])
>>> fuel.days is dep.metadata['days']
True
```

All of the variables present in the depletion file for this material are present, stored in the `data` dictionary. A few properties commonly used are accessible as attributes as well.

```
>>> fuel.data.keys()
dict_keys(['a', 'adens', 'burnup', 'gsrc', ..., 'volume'])
>>> fuel.adens
array([[  0.00000000e+00,   2.43591000e-09,   4.03796000e-09, ...,
          4.70133000e-09,   4.70023000e-09,   4.88855000e-09],
       [  0.00000000e+00,   6.06880000e-09,   8.11783000e-09, ...,
          8.05991000e-09,   8.96359000e-09,   9.28554000e-09],
       [  4.48538000e-06,   4.48486000e-06,   4.48432000e-06, ...,
          4.44726000e-06,   4.44668000e-06,   4.44611000e-06],
       ...,
       [  0.00000000e+00,   3.03589000e-11,   7.38022000e-11, ...,
          1.62829000e-09,   1.63566000e-09,   1.64477000e-09],
       [  0.00000000e+00,   1.15541000e-14,   2.38378000e-14, ...,
          8.60736000e-13,   8.73669000e-13,   8.86782000e-13],
       [  6.88332000e-02,   6.88334000e-02,   6.88336000e-02, ...,
          6.88455000e-02,   6.88457000e-02,   6.88459000e-02]])
```

Similar to the original file, the rows of the matrix correspond to positions in the isotopic vector, and the columns correspond to positions in burnup/day vectors.

```
>>> fuel.mdens.shape  # rows, columns
(34, 72)
>>> fuel.burnup.shape
(72,)
>>> len(fuel.names)
34
```

Data Retrieval

At the heart of the *DepletedMaterial* is the *getValues()* method. This method acts as an slicing mechanism that returns data for a select number of isotopes at select points in time.

```
>>> dayPoints = [0, 5, 10, 30]
>>> iso = ['Xe135', 'U235']
>>> vals = fuel.getValues('days', 'a', dayPoints, iso)
>>> print(vals.shape)
(2, 4)
>>> print(vals)
[[  0.00000000e+00   3.28067000e+14   3.24606000e+14   3.27144000e+14]
 [  5.36447000e+07   5.34519000e+07   5.32499000e+07   5.24766000e+07]]
```

The *DepletedMaterial* uses this slicing for the built-in *plot()* method

```
>>> fuel.plot('days', 'ingTox', dayPoints, iso,
              ylabel='Ingenstion Toxicity');
```
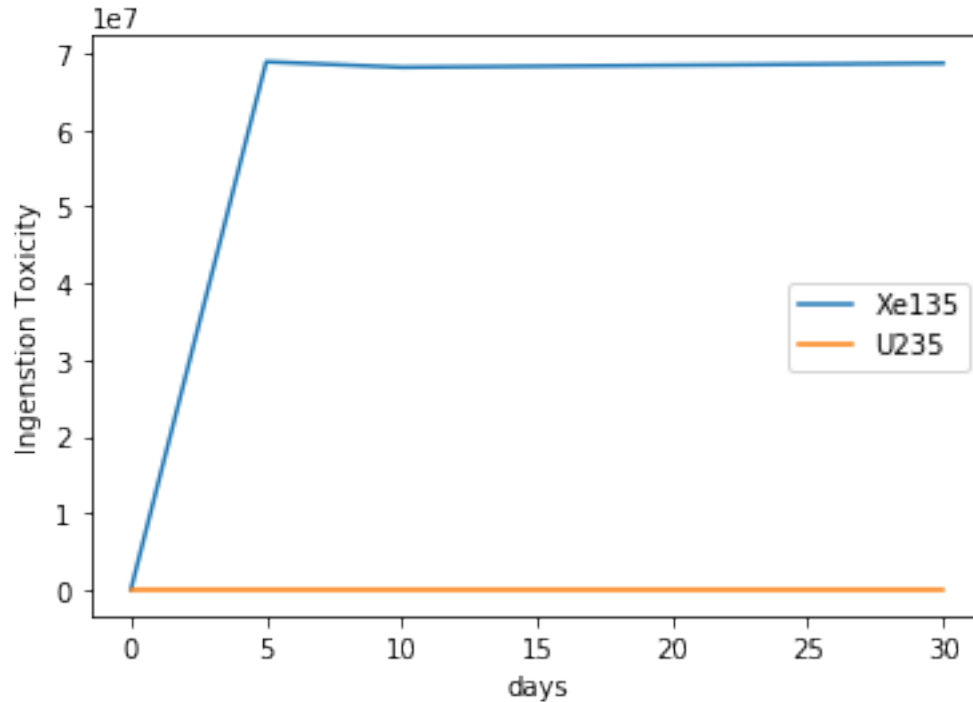
Limitations

Currently, the `DepletionReader` cannot catch materials with underscore in the name, due to variables like `ING_TOX` also containing an underscore. Issue #58

Settings

The `DepletionReader` also has a collection of settings to control what data is stored. If none of these settings are modified, the default is to store all the data from the output file.

```
>>> from serpentTools.settings import rc, defaultSettings
>>> for setting in defaultSettings:
>>>     if 'depletion' in setting:
>>>         print(setting)
>>>         for k, v in six.iteritems(defaultSettings[setting]):
>>>             print('\t', k, v)
depletion.materials
    type <class 'list'>
    description Names of materials to store. Empty list -> all materials.
    default []
depletion.processTotal
    type <class 'bool'>
    description Option to store the depletion data from the TOT block
    default True
depletion.materialVariables
    type <class 'list'>
    description Names of variables to store. Empty list -> all variables.
    default []
depletion.metadataKeys
    type <class 'list'>
```

```
    description Non-material data to store, i.e. zai, isotope names, burnup schedule, etc.
    options default
    default ['ZAI', 'NAMES', 'DAYS', 'BU']
```

Below is an example of configuring a `DepletionReader` that only stores the burnup days, and atomic density for all materials that begin with `bglass` followed by at least one integer.

Note: Creating   the   `DepletionReader`   in   this   manner   is   functionally   equivalent   to
`serpentTools.read(depFile)`

```
>>> rc['depletion.processTotal'] = False
>>> rc['depletion.metadataKeys'] = ['BU']
>>> rc['depletion.materialVariables'] = ['ADENS']
>>> rc['depletion.materials'] = [r'bglass\d+']
>>>
>>> bgReader = serpentTools.parsers.DepletionReader(depFile)
>>> bgReader.read()
INFO    : serpentTools: Preparing to read demo_dep.m
INFO    : serpentTools: Done reading depletion file
>>> bgReader.materials
{'bglass0': <serpentTools.objects.materials.DepletedMaterial at 0x239057dcb00>}
>>> bglass = bgReader.materials['bglass0']
>>> bglass.data
{'adens': array([[ 0.       , 0.       , 0.       , ..., 0.       , 0.       ,
         0.       ],
       [ 0.       , 0.       , 0.       , ..., 0.       , 0.       ,
         0.       ],
       [ 0.       , 0.       , 0.       , ..., 0.       , 0.       ,
         0.       ],
       ...,
       [ 0.       , 0.       , 0.       , ..., 0.       , 0.       ,
         0.       ],
       [ 0.       , 0.       , 0.       , ..., 0.       , 0.       ,
         0.       ],
       [ 0.0715841, 0.0715843, 0.0715845, ..., 0.0715968, 0.0715969,
         0.0715971]])}
>>> bglass.data.keys()
dict_keys(['adens'])
```

Conclusion

The `DepletionReader` is capable of reading and storing all the data from the SERPENT burned materials file. Upon reading, the reader creates custom `DepletedMaterial` objects that are responsible for storing and retrieving the data. These objects also have a handy `plot` method for quick analysis. Use of the `rc` settings control object allows increased control over the data selected from the output file.

References

1. J. Leppänen, M. Pusa, T. Viitanen, V. Valtavirta, and T. Kaltiaisenaho. "The Serpent Monte Carlo code: Status, development and applications in 2013." Ann. Nucl. Energy, 82 (2015) 142-150

DEVELOPER'S GUIDE

Contributing

First, thanks for your interest in contributing to this project! This document should help us expedite the process of reviewing issues and pull requests. For a quick look at all the issues that are up for grabs, take a look at the current unclaimed issues. If you claim an issue, use the `Assignees` setting to let us know that you've got it!

Scope

The scope of this project is to simplify and expedite analysis stemming from `SERPENT` outputs. In the future we may expand this project to expand to interacting heavily with input files, but that is currently beyond the scope of this project. Any and all issues, features and pull requests will be examined through this scope.

Issues

The goal for this project is to become the de facto method for processing `SERPENT` outputs and, if you're looking at this, there is some way we can improve. The GitHub issue tracker is the preferred way to post bug reports and feature requests.

Bug Reports

The more information given, the quicker we can reproduce and hopefully resolve the issue. Please see issue-template for a template that should be used for reporting bugs. One of the developers will add a bug label and start moving to resolve the issue posthaste.

Feature Requests

We are very interested in adding functionality from the SERPENT community! Requests can be done through the issue tracker as well. You can create an issue on the issue tracker with `[Feature]` or `[Request]` in the title. Describe what you would like to add, some expected results, and the purpose behind the feature. The development team will apply an enhancement label and proceed accordingly.

Pull Requests

Pull requests are how we review, approve, and incorporate changes into the `master` branch. If you have code you want to contribute, please look at the content in the *Developer's Guide* for things like *Pull Request Checklist*, *Coding Style*, and more.

When your content is ready for the pull request, follow the pull-request-template and make a request! Someone of the core development team will review the changes according to the criteria above and make changes and/or approve for merging!

Coding Style

For the most part, this project follows the **PEP 8** standard with a few differences. Some points are included here

- 79 characters per line

- Four spaces per indentation level

- Avoiding extraneous whitespace:

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )
```

Some of the specific style points for this project are included below

- mixedCase for variables, methods, and functions

- CamelCase for classes:

```
class DemoClass(object):

    def doSomething(self, arg0, longerArgumentName):
        pass
```

- Directly call the __init__ method from a parent class, e.g.:

```
class MyQueue(list):

    def __init__(self, items):
        list.__init__(self)
        self.extend(items)
```

- Arrange imports in the following order:

  1. imports from the standard library: os, sys, etc.

  2. imports from third party code: numpy, matplotlib, etc.

  3. imports from the serpentTools package

- Longer import paths are preferred to shorter:

```
# yes
from really.long.path.to.a import function
function()
# not preferred
import really
really.long.path.to.a.function()
```

Documentation

All public functions, methods, and classes should have adequate documentation through docstrings, examples, or inclusion in the appropriate file in the docs directory. Good forethought into documenting the code helps resolve issues and, in the case of docstrings, helps produce a full manual.

Docstrings

Docstrings are defined in **PEP 257** and are characterized by `"""triple double quotes"""`. These can be used to reduce the effort in creating a full manual, and can be viewed through python consoles to give the user insight into what is done, what is required, and what is returned from a particular object. This project uses numpy style docstrings, examples of which are given below

Functions and Methods

Below is the `depmtx()` function annotated using short and longer docstrings:

```python
def depmtx(filePath):
    """Return t, no, zai, a, and n1 values from the depmtx file."""
```

or:

```python
def depmtx(filePath):
    """
    Read the contents of the ``depmtx`` file and return contents

    .. note::

        If ``scipy`` is not installed, matrix ``A`` will be full.
        This can cause some warnings or errors if sparse or
        non-sparse solvers are used.

    Parameters
    ----------
    fileP: str
        Path to depletion matrix file

    Returns
    -------
    t: float
        Length of time
    n0: numpy.ndarray
        Initial isotopic vector
    zai: numpy.array
        String identifiers for each isotope in ``n0`` and ``n1``
    a: numpy.array or scipy.sparse.csc_matrix
        Decay matrix. Will be sparse if scipy is installed
    n1: numpy.array
        Final isotopic vector
    """
```

Both docstrings indicate what the function does, and what is returned. The latter, while more verbose, is preferred in most cases, for the following reasons. First, far more information is yielded to the reader. Second, the types of the inputs and outputs are given and clear. Some IDEs can obtain the expected types from the docstrings and inform the user if they are using an incorrect type.

More content can be added to the docstring, including

- Raises - Errors/warnings raised by this object

- See Also - follow up information that may be useful

- Yields - If this object is a generator. Similar to Returns

Classes

Classes can have a more lengthy docstring, as they often include attributes to which the class has access. Below is an example of the docstring for the `DepletionReader`:

```
"""
Parser responsible for reading and working with depletion files.

Parameters
----------
filePath: str
    path to the depletion file

Attributes
----------
materials: dict
    Dictionary with material names as keys and the corresponding
    :py:class:`~serpentTools.objects.DepletedMaterial` class
    for that material as values
metadata: dict
    Dictionary with file-wide data names as keys and the
    corresponding dataas values, e.g. 'zai': [list of zai numbers]
settings: dict
    names and values of the settings used to control operations
    of this reader

"""
```

Class docstrings can be added to the class signature, or to the `__init__` method, as:

```
class Demo(object):
    """
    Demonstration class

    Parameters
    ----------
    x: str
        Just a string

    Attributes
    ----------
    capX: str
        Capitalized x
    """
```

or:

```
def __init__(self, x):
    """
    Demonstration class

    Parameters
    ----------
    x: str
        Just a string

    Attributes
    ----------
    capX: str
```

```
        Capitalized x
    """
```

Deprecation

If an object is deprecated or will be modified in future versions, then the *deprecated()* and *willChange()* decorators should be applied to the object, and a note should be added to the docstring indicating as much.

Examples

When possible, features should be demonstrated, either through Jupyter notebooks in the `examples/` directory, or with an `Examples` section in the docstring. Specifically, all readers should be demonstrated as Jupyter notebooks that detail the typical usage, user control settings, and examples of how the data is stored and accessed.

Data Model

This project has two key offerings to the `SERPENT` community.

1. Scripting tools that are capable of interpreting the output files

2. Purpose-built containers for storing, accessing, and utilizing the data in the output files

Readers

The readers are the core of this package, and should have a logical and consistent method for structuring data. Each reader contained in this project should:

1. Be capable of reading the intended output file

2. Yield to the user the data in a logical manner

3. Default to storing all the data without user input

4. Receive input from the user regarding what data to restrict/include

5. Report activity at varying levels, including failure, to the user

The third and fourth points refer to use of the `serpentTools.settings` module to receive user input. The final point refers to use of the logging and messaging functions included in *serpentTools.messages*.

For example, the `DepletionReader` stores data in the following manner:

```
Reader
| - filePath
| - fileMetadata
    | - isotope names
    | - isotope ZAIs
    | - depletion schedule in days and burnup
| - materials
# for each material:
    | - DepletedMaterial-n
        | - name
        | - atomic density
        | - mass density
```

```
        | - volume
        | etc
```

Here, the reader primarily creates and stores material objects, and the useful data is stored on these objects. Some files, like the fission matrix and depletion matrix files may not have a structure that naturally favors this object-oriented approach.

Containers

The readers are primarily responsible for creating and populating containers, that are responsible for storing and retrieving data. Just like the `DepletedMaterial` has shortcuts for analysis like `getValues()` and `plot()`, each such container should easily and naturally provide access to the data, and also perform some common analysis on the data.

Pull Request Checklist

Below is the criteria that will be used in the process of reviewing pull requests (PR):

1. The content of the PR fits within the scope of the project - *Scope*

2. The code included in the PR is written in good pythonic fashion, and follows the style of the project - *Coding Style*

3. The code directly resolves a previously raised issue - *Issues*

4. PR does not cause unit tests and builds to fail

5. Changes are reflected in documentation - *Documentation*

API

Settings

class `serpentTools.settings.`**`UserSettingsLoader`**
> Class that stores the active user settings.

> **`expandVariables`**`()`
>> Extend the keyword groups into lists of serpent variables.

>>> **Returns** Names of all variables to be scraped

>>> **Return type** set

> **`getReaderSettings`**(*settingsPreffix*)
>> Get all module-wide and reader-specific settings.

>>> **Parameters** **`settingsPreffix`** (`str or list`) – Name of the specific reader. Will look for settings that lead with `readerName`, e.g. `depletion.metadataKeys` or `xs.variables`

>>> **Returns** Single level dictionary with `settingName:` `settingValue` pairs

>>> **Return type** dict

>>> **Raises** `KeyError` – If the reader name is not located in the `readers` settings dictionary

> **`loadYaml`**(*filePath*, *strict=True*)
>> Update the settings based on the contents of the yaml file

>> New in version 0.2.0.

>>> **Parameters**

>>> - **`filePath`** (`str`) – Path to config file

>>> - **`strict`** (`bool`) – Fail at the first incorrect setting. If false, failed settings will not be loaded and alerts will be raised

>>> **Raises**

>>> - *KeyError or TypeError* – If settings found in the config file are not valid

>>> - *FileNotFound or OSError* – If the file does not exist

> **`setValue`**(*name*, *value*)
>> Set the value of a specific setting.

>>> **Parameters**

>>> - **name** (`str`) – Full name of the setting

- **value** (*value to be set*) –

Raises

- KeyError – If the value is not one of the allowable options or if the setting does not match an existing setting

- TypeError – If the value is not of the correct type

Messages

System-wide methods for producing status update and errors.

See also:

- https://docs.python.org/2/library/logging.html

- https://www.python.org/dev/peps/pep-0391/

exception serpentTools.messages.**SerpentToolsException**
: Base-class for all exceptions in this project

serpentTools.messages.**critical**(*message*)
: Log that something has gone horribly wrong.

serpentTools.messages.**debug**(*message*)
: Log a debug message.

serpentTools.messages.**deprecated**(*useInstead*)
: Decorator that warns that different function should be used instead.

serpentTools.messages.**error**(*message*)
: Log that something caused an exception but was supressed.

serpentTools.messages.**info**(*message*)
: Log an info message, e.g. status update.

serpentTools.messages.**updateLevel**(*level*)
: Set the level of the logger.

serpentTools.messages.**warning**(*message*)
: Log a warning that something that could go wrong or should be avoided.

serpentTools.messages.**willChange**(*changeMsg*)
: Decorator that warns that some functionality may change.

Parser Module

The main module associated with reading SERPENT files. For information on each individual reader, please see the dedicated file listed below

Reader	Hyperlink
Branching	*Branching Reader*
Bumat	*Bumat Reader*
Depletion	*Depletion Reader*
Detector	*Detector Reader*
FissionMatrix	*Fission Matrix Reader*
Results	*Results Reader*

serpentTools.parsers.**inferReader**(*filePath*)

> Attempt to infer the correct reader type.

>> **Parameters** **filePath** (`str`) – File to be read.

>> **Raises** `SerpentToolsException` – If a reader cannot be inferred

serpentTools.parsers.**read**(*filePath*, *reader='infer'*)

> Simple entry point to read a file and obtain the processed reader.

> **Note:** If you know the type of reader you will be using, it is best to either pass in the string argument, or directly use the appropriate reader class

---

>> **Parameters**

>> - **filePath** (`str`) – Path to the file to be reader
>> - **reader** (`str or callable`) – Type of reader to use. If a string is given, then the actions described below will happen. If callable, then that function will be used with the file path as the first argument.

String argument	Action
infer	Infer the correct reader based on the file
dep	`DepletionReader`
branch	`BranchingReader`
det	`DetectorReader`
results	`ResultsReader`
bumat	`BumatReader`
fission	`FissionMatrixReader`

>> **Returns** Correct subclass corresponding to the file type

>> **Return type** serpentTools.objects.readers.BaseReader

>> **Raises**

>> - `AttributeError` – If the object created by the reader through `reader(filePath)` does not have a `read` method.
>> - `SerpentToolsException` – If the reader could not be inferred or if the requested reader string is not supported
>> - `NotImplementedError` – This has the ability to load in readers that may not be complete, and thus the `read` method may raise this error.

serpentTools.parsers.**depmtx**(*fileP*)

> Read the contents of the `depmtx` file and return contents

> **Note:** If `scipy` is not installed, matrix `A` will be full. This can cause some warnings or errors if sparse or non-sparse solvers are used.

>> **Parameters** **fileP** (`str`) – Path to depletion matrix file

>> **Returns**

>> - **t** (*float*) – Length of time
>> - **n0** (*numpy.ndarray*) – Initial isotopic vector

- **zai** (*numpy.array*) – String identifiers for each isotope in `n0` and `n1`

- **a** (*numpy.array or scipy.sparse.csc_matrix*) – Decay matrix. Will be sparse if scipy is installed

- **n1** (*numpy.array*) – Final isotopic vector

Parsing Engines

The classes contained in `serpentTools/engines.py` are part of the [drewtils](#) v0.1.9 package and are provided under the following license:

The MIT License (MIT)

Copyright (c) Andrew Johnson, 2017

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

These are designed to facilitate the parsing of files with a regular structure. For example, the depletion files all contain "chunks" of data that are separated by empty lines. Each chunk leads off with either the name of the material and associated variable, or the metadata, e.g. `ZAI`, `DAYS`. These parsers help break up these files into more digestible pieces.

Note: For developers, it is not required that these classes be used. These are bundled with this project to eliminate the need to install extra packages. Some of the readers, like the `BranchingReader` are not well suited for this type of parsing.

---

class `serpentTools.engines.`**`KeywordParser`** (*filePath*, *keys*, *separators=None*, *eof=''*)

Class for parsing a file for chunks separated by various keywords.

> **Parameters**
>
> - **`filePath`** (`str`) – Object to be read. Any object with `read` and `close` methods
>
> - **`keys`** (`Iterable`) – List of keywords/phrases that will indicate the start of a chunk
>
> - **`separators`** (`Iterable or None`) – List of additional phrases that can separate two chunks. If not given, will default to empty line `'\n'`.
>
> - **`eof`** (`str`) – String to indicate the end of the file

`line`
> *str*
>
> Most recently read line

parse()
>   Parse the file and return a list of keyword blocks.

>   >   **Returns** List of key word argument chunks.

>   >   **Return type** list

yieldChunks()
>   Return each chunk of text as a generator.

>   >   **Yields** *list* – The next chunk in the file.

class serpentTools.engines.**PatternReader**(*filePath*)
>   Class that can read over a file looking for patterns.

>   >   **Parameters** **filePath**(*str*) – path to the file that is to be read

line
>   *str*

>   Most recently read line

match
>   *regular expression match or None*

>   Match from the most recently read line

searchFor(*pattern*)
>   Return true if the pattern is found.

>   >   **Parameters** **pattern**(*str or compiled regular expression*) –

>   >   **Returns** bool

>   >   **Return type** True if the pattern was found

yieldMatches(*pattern*)
>   Generator that returns all match groups that match pattern.

>   >   **Parameters** **pattern**(*str or compiled regular expression*) – Seek through the
>   >   file and yield all match groups for lines that contain this patten.

>   >   **Yields** *sequential match groups*

Containers

Many of the readers utilize custom built objects for storing data in a logical and accessible format. These containers
are detailed here.

Note: Variables taken from SERPENT will be converted to mixedCase and stored under those names. for example,
INF_TOT would be stored as infTot

---

class serpentTools.objects.containers.**BranchContainer**(*parser*, *branchID*, *branchNames*,
>   >   >   >   >   >   >   >   >   >   >   >   >   >   >   *stateData*)
>   Class that stores data for a single branch.

>   The [*BranchingReader*](#) stores branch variables and branched group constant data inside these container
>   objects. These are used in turn to create [*HomogUniv*](#) objects for storing group constant data.

>   >   **Parameters**

- **parser** (*serpentTools.objects.readers.BaseReader*) – Parser that read the file that created this object

- **branchID** (*int*) – Index for the run for this branch

- **branchNames** (*tuple*) – Name of branches provided for this universe

- **stateData** (*dict*) – key: value pairs for branch variables

stateData
> *dict*

> Name: value pairs for the variables defined on each branch card

universes
> *dict*

> Dictionary storing the homogenized universe objects. Keys are tuples of (`universeID`, `burnup`, `burnIndex`)

addUniverse(*univID*, *burnup=0*, *burnIndex=0*, *burnDays=0*)
> Add a universe to this branch.

> Data for the universes are produced at specific points in time. The additional arguments help track when the data for this universe were created.

> **Warning:** This method will overwrite data for universes that already exist

> **Parameters**
>
> - **univID** (*int or str*) – Identifier for this universe
>
> - **burnup** (*float or int*) – Value of burnup [MWd/kgU]
>
> - **burnIndex** (*int*) – Point in the depletion schedule
>
> - **burnDays** (*int or float*) – Point in time
>
> **Returns** newUniv
>
> **Return type** *serpentTools.objects.containers.HomogUniv*

getUniv(*univID*, *burnup=None*, *index=None*)
> Return a specific universe given the ID and time of interest

> If burnup and index are given, burnup is used to search

> **Parameters**
>
> - **univID** (*int*) – Unique ID for the desired universe
>
> - **burnup** (*float or int*) – Burnup [MWd/kgU] of the desired universe
>
> - **index** (*int*) – Point of interest in the burnup index
>
> **Returns** univ – Requested Universe
>
> **Return type** *serpentTools.objects.containers.HomogUniv*
>
> **Raises**
>
> - KeyError: – If the requested universe could not be found
>
> - SerpentToolsException: – If neither burnup nor index are given

orderedUniv
> Universe keys sorted by ID and by burnup

class serpentTools.objects.materials.**DepletedMaterial**(*parser*, *name*)
    Class for storing material data from `_dep.m` files.

> **Parameters**
>
> - **parser** (*DepletionReader*) – Parser that found this material. Used to obtain file meta-data like isotope names and burnup
>
> - **name** (*str*) – Name of this material

zai
    numpy.array or None

    Isotope id's

names
    numpy.array or None

    Names of isotopes

days
    numpy.array or None

    Days overwhich the material was depleted

adens
    numpy.array or None

    Atomic density over time for each nuclide

mdens
    numpy.array or None

    Mass density over time for each nuclide

burnup
    numpy.array or None

    Burnup of the material over time

addData(*variable*, *rawData*)
    Add data straight from the file onto a variable.

> **Parameters**
>
> - **variable** (*str*) – Name of the variable directly from `SERPENT`
>
> - **rawData** (*list*) – List of strings corresponding to the raw data from the file

getValues(*xUnits*, *yUnits*, *timePoints=None*, *names=None*)
    Return x values for given time, and corresponding isotope values.

> **Parameters**
>
> - **xUnits** (*str*) – name of x value to obtain, e.g. `'days'`, `'burnup'`
>
> - **yUnits** (*str*) – name of y value to return, e.g. `'adens'`, `'burnup'`
>
> - **timePoints** (*list or None*) – If given, select the time points according to those specified here. Otherwise, select all points
>
> - **names** (*list or None*) – If given, return y values corresponding to these isotope names. Otherwise, return values for all isotopes.
>
> **Returns** Array of values.
>
> **Return type** numpy.array

Raises

- `AttributeError` – If the names of the isotopes have not been obtained and specific isotopes have been requested

- `KeyError` – If at least one of the days requested is not present

getXY(**args*, ***kwargs*)
  Return x values for given time, and corresponding isotope values.

  Deprecated since version 0.1.0: Use `getValues()` instead.

plot(*xUnits*, *yUnits*, *timePoints=None*, *names=None*, *ax=None*, *legend=True*, *label=True*, *xlabel=None*, *ylabel=None*)
  Plot some data as a function of time for some or all isotopes.

  Parameters

- **xUnits** (`str`) – name of x value to obtain, e.g. `'days'`, `'burnup'`

- **yUnits** (`str`) – name of y value to return, e.g. `'adens'`, `'burnup'`

- **timePoints** (`list or None`) – If given, select the time points according to those specified here. Otherwise, select all points

- **names** (`list or None`) – If given, return y values corresponding to these isotope names. Otherwise, return values for all isotopes.

- **ax** (None or `matplotlib axes`) – If given, add the data to this plot. Otherwise, create a new plot

- **legend** (`bool`) – Automatically add the legend

- **label** (`bool`) – Automatically label the axis

- **xlabel** (`None or str`) – If given, use this as the label for the x-axis. Otherwise, use xUnits

- **ylabel** (`None or str`) – If given, use this as the label for the y-axis. Otherwise, use yUnits

  Returns Axes corresponding to the figure that was plotted

  Return type `matplotlib axes`

  See also:

  [`getValues()`](#)

class serpentTools.objects.containers.**Detector**(*parser*, *name*)
  Class to store detector data.

  Parameters

- **parser** ([`DetectorReader`](#)) – Detector reader that created this detector

- **name** (`str`) – Name of this detector

bins
  numpy.array

  Tallies straight from the detector file

grids
  dict

  Dictionary with additional data describing energy grids or mesh points

tallies
> *None or numpy.array*

> Reshaped tally data to correspond to the bins used

errors
> *None or numpy.array*

> Reshaped relative error data corresponsing to bins used

scores
> *None or numpy.array*

> Reshaped array of tally scores. SERPENT 1 only

indexes
> *None or OrderedDict*

> Collection of unique indexes for each requested bin

addTallyData(*bins*)
> Add tally data to this detector

meshPlot(*xdim*, *ydim*, *what='tallies'*, *fixed=None*, *ax=None*, *cmap=None*, *addcbar=True*, *xlabel=None*, *ylabel=None*, *xscale='linear'*, *yscale='linear'*)
> Plot tally data as a function of two mesh dimensions

> **Parameters**

> - **xdim** (`str`) – Primary dimension - will correspond to x-axis on plot

> - **ydim** (`str`) – Secondary dimension - will correspond to y-axis on plot

> - **what** (`{'tallies', 'errors', 'scores'}`) – Color meshes from tally data, uncertainties, or scores

> - **fixed** (`None or dict`) – Dictionary controlling the reduction in data down to one dimension

> - **ax** (`axes or None`) – Axes on which to plot the data

> - **cmap** (`None or str`) – Colormap to apply to the figure. If None, use default colormap

> - **addcbar** (`bool`) – If True, add a colorbar to the figure

> - **xlabel** (`None or str`) – Label to apply to x-axis. If not given, defaults to xdim

> - **ylabel** (`None or str`) – Label to apply to y-axis. If not given, defaults to xdim

> - **xscale** (`{'log', 'linear'}`) – Scale to apply to x-axis

> - **yscale** (`{'log', 'linear'}`) – Scale to apply to y-axis

> **Returns** **ax** – Ax on which the data was plotted

> **Return type** pyplot.Axes

> **Raises**

> - `SerpentToolsException` – If data to be plotted, with or without constraints, is 2D

> - `KeyError` – If the data set by `what` not in the allowed selection

> See also:

> [*slice()*](#)

plot (*xdim=None*, *what='tallies'*, *sigma=None*, *fixed=None*, *ax=None*, *xlabel=None*, *ylabel=None*, *steps=False*, ***kwargs*)

Shortcut routine for plotting 1D data

Parameters

- **xdim** (`None or str`) – If not None, use the array under this key in `indexes` as the x axis

- **what** (`{'tallies', 'errors', 'scores'}`) – Primary data to plot

- **sigma** (`None or int`) – If given, apply this level of confidence to absolute errors. If not given, then error bars will not be used

- **fixed** (`None or dict`) – Dictionary controlling the reduction in data down to one dimension

- **ax** (`axes or None`) – Axes on which to plot the data

- **xlabel** (`None or str`) – If given, apply this label to the x-axis. If `xdim` is given and `xlabel` is None, then `xdim` will be applied to the x-axis

- **ylabel** (`None or str`) – If given, label to apply to y-axis

- **steps** (`bool`) – If true, plot the data as constant inside the respective bins. Sets `drawstyle` to be `steps-post` unless `drawstyle` given in `kwargs`

- **kwargs** (`dict`) – additional arguments to pass to the `plot()` of `errorbar()` function.

Returns ax

Return type matplotlib.AxesSubplot

Raises SerpentToolsException: If the data is not constrained to 1D

See also:

- *slice()*

- *spectrumPlot()* better options for plotting energy spectra

reshape ()

Reshape the tally data into a multidimensional array

This method reshapes the tally and uncertainty data into arrays where the array axes correspond to specific bin types. If a detector was set up to tally two group flux in a 5 x 5 xy mesh, then the resulting tally data would be in a 50 x 12/13 matrix in the original `detN.m` file. The tally data and relative error would be rebroadcasted into 2 x 5 x 5 arrays, and the indexing information is stored in `self.indexes`

Returns shape – Dimensionality of the resulting array

Return type list

Raises SerpentToolsException: – If the bin data has not been loaded

slice (*fixed*, *data='tallies'*)

Return a slice of the reshaped array where certain axes are fixed

Parameters

- **data** (`{'tallies', 'errors', 'scores'}`) – Which data set to slice

- **fixed** (`dict`) – dictionary to aid in the restriction on the multidimensional array. Keys correspond to the various grids present in `indexes` while the values are used to

Returns **reduced** – View into the respective data where certain dimensions have been removed

Return type numpy.array

Raises

- `SerpentToolsException` – If the data has not been reshaped or is None [e.g. scores]

- `KeyError` – If the data set to slice not in the allowed selection

spectrumPlot (*fixed=None*, *ax=None*, *normalize=True*, *xlabel=None*, *ylabel=None*, *steps=True*, *xscale='log'*, *yscale='linear'*, *sigma=3*, ***kwargs*)
Quick plot of the detector value as a function of energy.

Parameters

- **fixed** (`None or dict`) – Dictionary controlling the reduction in data down to one dimension

- **ax** (`pyplot.Axes or None`) – Ax on which to plot the data

- **normalize** (`bool`) – Normalize quantities per unit lethargy

- **xlabel** (`str or None`) – Label for x-axis. Defaults to 'Energy [MeV]'

- **ylabel** (`str or None`) – Label for y axis

- **steps** (`bool`) – Plot tally as constant inside bin

- **xscale** (`{'log', 'linear'}`) – Scale to apply to x axis

- **yscale** (`{'log', 'linear'}`) – Scale to apply to y axis

- **sigma** (`int`) – Level of confidence to apply to errors. Use for no error bars

- **kwargs** – Additional arguments to pass to plot command

Returns **ax** – Axes on which the data was plotted

Return type pyplot.Axes

Raises `SerpentToolsException` – if number of rows in data not equal to number of energy groups

See also:

[*slice()*](#)

class serpentTools.objects.containers.**HomogUniv** (*container*, *name*, *bu*, *step*, *day*)
Class for storing homogenized universe specifications and retrieving them

Parameters

- **container** (`serpentTools.objects.readers.BaseReader or`) – serpentTools.objects.containers.BranchContainer Object to which this universe is attached

- **name** (`str`) – name of the universe

- **bu** (`float`) – burnup value

- **step** (`float`) – temporal step

- **day** (`float`) – depletion day

name
str

name of the universe

bu
> *float*

> burnup value

step
> *float*

> temporal step

day
> *float*

> depletion day

infExp
> *dict*

> Expected values for infinite medium group constants

infUnc
> *dict*

> Relative uncertainties for infinite medium group constants

b1Exp
> *dict*

> Expected values for leakage corrected group constants

b1Unc
> *dict*

> Relative uncertainties for leakage-corrected group constants

metadata
> *dict*

> Other values that do not not conform to inf/b1 dictionaries

addData(*variableName*, *variableValue*, *uncertainty=False*)
> Sets the value of the variable and, optionally, the associate s.d.

> > **Warning:** This method will overwrite data for variables that already exist

> > **Parameters**
> >
> > - **variableName** (`str`) – Variable Name
> >
> > - **variableValue** – Variable Value
> >
> > - **uncertainty** (`bool`) – Set to `True` in order to retrieve the uncertainty associated to the expected values
> >
> > **Raises** `TypeError` – If the uncertainty flag is not boolean

get(*variableName*, *uncertainty=False*)
> Gets the value of the variable VariableName from the dictionaries

> > **Parameters**
> >
> > - **variableName** (`str`) – Variable Name
> >
> > - **uncertainty** (`bool`) – Boolean Variable- set to True in order to retrieve the uncertainty associated to the expected values
> >
> > **Returns**

- *x* – Variable Value

- *dx* – Associated uncertainty

Raises

- `TypeError` – If the uncertainty flag is not boolean

- `KeyError` – If the variable requested is not stored on the object

Branching Reader

The *BranchingReader* is designed to read the files generated by the automated burnup sequence. The output format is described at the SERPENT Wiki.

The reader stores homogenized universe data for each universe at each point in time for each branch in *HomogUniv* objects.

class `serpentTools.parsers.branching.``**BranchingReader**`(*filePath*)
>   Parser responsible for reading and working with automated branching files.

>   >   **Parameters** `filePath` (`str`) – path to the depletion file

>   `iterBranches`()
>   >   Iterate over branches yielding paired branch IDs and containers

>   `read`()
>   >   Read the branching file and store the coefficients.

Bumat Reader

> **Warning:** This reader is not implemented yet. This merely serves as a placeholder

class `serpentTools.parsers.bumat.``**BumatReader**`(*filePath*, *readerSettingsLevel*)
>   Parser responsible for reading and working with burned material files.

>   >   **Parameters** `filePath` (`str`) – path to the depletion file

Depletion Reader

> **Warning:** Does not support depleted materials with underscores, i.e. `fuel_1` will not be matched with the current methods

class `serpentTools.parsers.depletion.``**DepletionReader**`(*filePath*)
>   Parser responsible for reading and working with depletion files.

>   >   **Parameters** `filePath` (`str`) – path to the depletion file

>   `materials`
>   >   *dict*

>   >   Dictionary with material names as keys and the corresponding `DepletedMaterial` class for that material as values

metadata
> *dict*
>
> Dictionary with file-wide data names as keys and the corresponding dataas values, e.g. 'zai': [list of zai numbers]

settings
> *dict*
>
> names and values of the settings used to control operations of this reader

__delattr__
> x.__delattr__('name') <==> del x.name

__format__ ()
> default object formatter

__getattribute__
> x.__getattribute__('name') <==> x.name

__hash__

__reduce__ ()
> helper for pickle

__reduce_ex__ ()
> helper for pickle

__repr__

__setattr__
> x.__setattr__('name', value) <==> x.name = value

__sizeof__ () → int
> size of object in memory, in bytes

read ()
> Read through the depletion file and store requested data.

Detector Reader

class `serpentTools.parsers.detector.`**`DetectorReader`** (*filePath*)
> Parser responsible for reading and working with detector files.
>
> > **Parameters** **filePath** (*str*) – path to the depletion file

read ()
> Read the file and store the detectors.

Fission Matrix Reader

> **Warning:** This reader is not implemented yet. This merely serves as a placeholder

class `serpentTools.parsers.fissionMatrix.`**`FissionMatrixReader`** (*filePath*, *readerSettingsLevel*)
> Parser responsible for reading and working with fission matrix files.
>
> > **Parameters** **filePath** (*str*) – path to the depletion file

Results Reader

> **Warning:** This reader is not implemented yet. This merely serves as a placeholder

class `serpentTools.parsers.results.`**`ResultsReader`**(*filePath*, *readerSettingsLevel*)

   Parser responsible for reading and working with result files.

   > **Parameters** **`filePath`** (*str*) – path to the depletion file

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

S