serpentTools Documentation

The serpentTools developer team

Feb 20, 2020

CONTENTS:

A suite of parsers designed to make interacting with *SERPENT* [serpent] 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 http://montecarlo.vtt.fi

The Annals of Nuclear Energy article should be cited for all work using *SERPENT*.

Preferred citation for attribution

Andrew Johnson, Dan Kotlyar, Gavin Ridley, Stefano Terlizzi, & Paul Romano. (2018, June 29). "serpent-tools: A collection of parsing tools and data containers to make interacting with SERPENT outputs easy, intuitive, and flawless.,".

CHAPTER

ONE

PROJECT OVERVIEW

The `serpentTools` package contains a variety of parsing utilities, each designed to read a specific output from the *SERPENT* Monte Carlo code [serpent]. Many of the parsing utilities store the outputs in custom container objects, while other store or return a collection of arrays. This page gives an overview of what files are currently supported, including links to examples and relevant documentation.

Unless otherwise noted, all the files listed below can be read using *`serpentTools.read()`*. For example:

```
>>> import serpentTools
>>> res = serpentTools.read('my_resFile_res.m')
```

would return a *`ResultsReader`*

Some file-types have an additional reader that is capable of reading multiple files and computing actual uncertainties. These `samplers` are detailed in *Samplers* and listed under the supported files.

Many of the readers have examples present in the *Examples* section. Each example is present as a *Jupyter notebook* at Github: examples. These can be used as a launching point for tutorials or for your own analysis.

1.1 Main output File

- File Description: Primary output file for transport calculation - `[input]_res.m`
- SERPENT wiki: result wiki
- Primary Reader: *`ResultsReader`*
- Example - notebook: ResultsReader
- Example - manual: *Results Reader*

1.2 Depletion File

- File Description: Output from burnup calculations, showing quantities for various materials - `[input]_dep.m`
- SERPENT wiki: depletion wiki
- Primary Reader: *`DepletionReader`*
- Example - notebook: DepletionReader
- Example - manual: *Depletion Reader*
- Sampler - *`serpentTools.samplers.DepletionSampler`*

1.3 Detector/ Tally File

- File Description: Output generated by creating detectors - `[input]_det[N].m`
- SERPENT wiki: detector definition
- Primary Reader: *DetectorReader*
- Example - notebook: Detector
- Example - manual: *Detector Reader*
- Sampler - *serpentTools.samplers.DetectorSampler*

1.4 Sensitivity File

- File Description: Output giving sensitivities of defined responses to perturbations - `[input]_sens[N].m`
- SERPENT wiki: sensitivity wiki
- Primary Reader: *SensitivityReader*
- Example - notebook: Sensitivity
- Example - manual: *Sensitivity Reader*

1.5 Branching Coefficient File

- File Description: Selected homogenized group constants across a variety of defined branch states - `[input].coe`
- SERPENT wiki: branching wiki
- Primary Reader: *BranchingReader*
- Secondary Read function: `serpentTools.BranchCollector.fromFile()`
- Example - notebook: Branching
- Example - manual: *Branching Reader*

1.6 Micro-depletion File

- File Description:
- SERPENT wiki: microxs wiki
- Primary Reader: *MicroXSReader*
- Example - notebook: MicroXSReader
- Example - manual: *Micro cross section reader*

1.7 Cross Section Plot File

- File Description: Cross section data and energy grids for various reactions - `[input]_xs0.m`
- SERPENT wiki: *xsplot wiki_*
- Primary Reader: *`XSPlotReader`*
- Example - notebook: XSPlot
- Example - manual: *Cross Section Reader/Plotter*

1.8 Depletion Matrix File

- File Description: Data pertaining to depletion for a single burnable material at a single point in time - `depmtx_[material-identifier]-s[step].m`
- Primary Reader: *`DepmtxReader`*
- Example - notebook: DepletionMatrix
- Example - manual: *Depletion Matrix Reader*

CHAPTER

TWO

INSTALLATION

In order to properly install this package, you need a version of python that plays well with libraries like *numpy* and *matplotlib*. This is easy or less easy depending on your operating system, but we have a brief walk through at *Obtaining a Python Distribution*.

2.1 Terminal terminology

Commands below that begin with `$` must be run from the terminal containing your preferred python distribution, neglecting the `$`. Depending on operating system, this could be the normal terminal, or a modified prompt. See *Obtaining a Python Distribution* for more information if you are not sure.

Commands that begin with `>>>` should be run inside of a python environment. The following would be a valid set of instructions to pass to your terminal, printing a very basic python command:

```
$ python -m "print('hello world')"
hello world
$ python
>>> print('hello world')
hello world
```

2.2 Installing from pip

pip is the easiest way to install the latest version of `serpentTools`. First, ensure that you have *numpy* installed, as this is one of the required packages that is tricky to install. You can check by opening up a your preferred terminal and executing

```
$ python -c "import numpy"
```

If this fails, please consult *Obtaining a Python Distribution*.

Next, installation with pip can be completed with:

```
$ python -m pip install --user --upgrade pip serpentTools
```

This installs the dependencies of the package, the latest `serpentTools` release, and upgrades your version of *pip* along the way. When a new release is issued, run the command again to install the updated version.

If you wish to install the development branch to get the latest updates as they happen, use the following command:

```
$ python -m pip install -e git+https://www.github.com/CORE-GATECH-GROUP/serpent-tools.
↪git@develop#egg=serpentTools
```

Note: Changes to the development branch may not always be back-compatible and may cause non-ideal outcomes. Be sure to check the *Changelog* before/after updating from the develop branch

See also pip install from git

2.3 Installing from a Release

1. Download the source code for the latest release from GitHub releases as a `.zip` or `.tar.gz` compressed file.
2. Extract/decompress the contents somewhere convenient and memorable
3. Open your terminal and navigate to this directory:

```
$ cd path/to/release
```

4. Install using our setup script:

```
$ python setup.py install
```

5. Verify the install by running our test suite:

```
$ python setup.py test
```

2.4 Installing via git

The newest features are available on the `develop` branch before getting pushed to the `master` branch. We try to give decent notice when features are going to change via warning messages and our *Changelog*, but changes to the API and other functionality can occur across the develop branch.

1. Clone the repository and checkout the branch of your choosing. The default is `develop`:

```
$ git clone https://github.com/CORE-GATECH-GROUP/serpent-tools.git
$ cd serpent-tools
$ git checkout master
```

2. Install using our setup script:

```
$ python setup.py install
```

3. Verify the install by running our test suite:

```
$ python setup.py test
```

2.5 Obtaining a Python Distribution

Obtaining a version of python into which `serpent-tools` can be installed varies by operating system, with Windows requiring the most finesse.

2.5.1 Linux/Mac/Unix-like Operating Systems

If you don't have *numpy* installed, you will have to obtain it from your package manager or from pip:

```
# ubuntu
$ sudo apt-get install python-numpy
# pip
$ sudo pip install --upgrade numpy
```

If you already have *numpy*, then the *pip* installtion process will take care of our other dependencies.

2.5.2 Windows

The easiest and most painless way to obtain packages like *numpy* on Windows is with either the *Anaconda* or *Miniconda* distributions. Each of these also includes the *Anaconda Prompt* which is a modified terminal that plays better with Python. The former comes with a few hundred packages, included most of the ones needed for this project, bundled for you. The latter is a very small distribution and requires you to install the packages you want via *conda*. Should you choose this route, then you need to launch the *Anaconda Prompt* and install with:

```
$ conda install setuptools numpy matplotlib pyyaml
```

This prompt is what you should use when following the instructions in in *Installation*.

CHAPTER

THREE

CHANGELOG

3.1 0.9.2

- Officially support installing under Python 3.8
- Support for passing threshold values to hexagonal detector plots - #351

3.1.1 Bug Fixes

- Detector reader can handle sequential detectors with very similar names - #374.
- `serpentTools` doesn't make any modifications to the logging state, other than introducing package-wide logger.
- Colorbars for mesh plots are placed next to their corresponding plot, rather than near the last drawn plot - #372

3.2 0.9.1

3.2.1 Bug Fixes

- Sensitivity arrays generated with `sens opt history 1` will no longer overwrite the primary result arrays - #366. These arrays are not currently stored - #367

3.3 0.9.0

- Python 2 support has been dropped.
- Add support for installing and testing against Python 3.7

3.4 0.8.1

- Use `six>=1.13.0`
- Use `yaml>=5.1.1`

3.4.1 Bug Fixes

- Fix *detector.names* setting

3.5 0.8.0

Warning: Serpent 1 detectors are no longer supported - #327. Version 0.9.0 will remove support for python 2 - #328

- Better handling of discontinuity factors - #329
- *HomogUniv* objects no longer automatically convert data to arrays
- Serpent 2.1.31 is the default version for *serpentVersion* setting
- Detectors and related subclasses are now standalone classes that can be imported as `serpentTools.Detector` - #341
- *BranchContainer* now inherits from `dict` - #344
- Keys for universes in `ResultsReader.universes` are `UnivTuple`
- Keys for microscopic cross sections in `MicroXSReader.xsVal` and `MicroXSReader.xsUnc` are `MicroXSTuple`
- Spread plots for sampled detector and depletion containers allow changing how the mean data and sampled data are plotted by passing dictionary of matplotlib commands, e.g. `meanKwargs={"c": "r", "marker": x"}` would plot the mean data in red with crosses as markers.

3.5.1 Bug Fixes

- Burnup and days are properly set on homogenized universes when reading a result file with multiple universes but no burnup - #346
- Modifications made to detector tally data will be reflected in later plots - #337, #341
- Variable groups for version 2.1.31 are properly expanded - #347

3.5.2 Incompatible API Changes

- Values are stored in array form on *HomogUniv* when it makes sense. For example, values like `infKinf` are stored as scalars.
- Setting `expectGcu` has been removed as #324 fixed how files without group constants are handled.
- Keys to *BranchedUniv* objects stored in *serpentTools.xs.BranchCollector.universes* are stored as strings, rather than integers, e.g. `0` is replaced with `"0"` - #321
- Keys to *HomogUniv* instances stored on *BranchContainer* are now `UnivTuple`, or tuples with `universe, burnup, step, days` - #344
- *serpentTools.Detector.indexes* is now a tuple of strings describing each dimension of `tallies` rather than `OrderedDict` - #341

3.6 0.7.1

- Add *__getitem__()* and *__setitem__()* convenience methods for accessing expected values on *HomogUniv* objects
- Add `thresh` argument to *Detector* `meshPlot` where only data greater than `thresh` is plotted.
- Mitigate pending deprecated imports from `collections` - #313
- Increase required version of *yaml* to `5.1.1`
- Include `SERPENT 2.1.31` support in *serpentVersion* setting

3.6.1 Bug fixes

- Tally data for detectors with time-bins are properly handled - #312
- Support for generic string universe names for *BranchingReader* and *BranchCollector* - #318

3.6.2 Pending Deprecations

- Keys to *BranchedUniv* objects stored in *serpentTools.xs.BranchCollector.universes* are stored as strings, rather than integers, e.g. `0` is replaced with `"0"`. A workaround is in-place, but will be removed in future versions.
- `SERPENT` 1 style detectors with additional score column will not be supported starting at version `0.8.0`.

3.7 0.7.0

- **Easier construction of *BranchCollector* objects - #276**
  - Directly from the class *fromFile*
  - Don't require passing branch information to *BranchCollector*. Will be inferred from file and set with `(p0, p1, ...)`. State data can be used to determine which index is a given perturbation type.
- Direct `toMatlab` methods for *ResultsReader*, *SensitivityReader*, *DepmtxReader* *DepletionReader*, *DetectorReader*, *HistoryReader*, and *Detector* objects - #290, #291

- Overhaul, reorganization, and cleanup of documentation

3.7.1 Incompatible API Changes

- *HomogUniv* objects are now stored on *ResultsReader* with zero-based indexing for burnup. The previous first value of burnup step was one. All burnup indices are now decreased by one. Similarly, if no burnup was present in the file, the values of burnup and days for all universes is zero - #288
- When reading Detectors with a single tally, the value of `tallies`, `errors`, and `scores` are stored as floats, rather than *numpy* arrays - #289

3.7.2 Deprecations

- *DepletionReader* `saveAsMatlab` in favor of *toMatlab()*
- SERPENT `2.1.30` is the default version of *serpentVersion*. Will alter some variable groups, like *optimization* and *optimization*, that exist in both versions but are slightly different.

3.7.3 Bug Fixes

- *BranchingReader* is now capable of reading `.coe` files with uncertainties - #272
- Fixed a bug that caused some plots not to return the axes object of the plot - #297
- *HomogUniv* plots are plotted against energy group when no group structure can be determined, and now labeled as such - #299
- Removed a non-zero exit code from a successful use of the *Random Seed Generation* command line command - #300
- *ResultsReader* can process files with assembly discontinuity factors (ADFs) - #305

3.8 0.6.2

- Data files are bundled in source distribution
- CLI interface for converting some output files to matlab files - *Conversion to Binary .mat files*
- Add `serpentTools.io` module for converting objects to other data types. Currently a general function for converting `toMatlab()`
- *DetectorReader* and *Detector* objects can be written to MATLAB files using `serpentTools.io.toMatlab()`
- *ResultsReader* can plot data using *plot()*
- Experimental *BranchCollector* for collecting group constants from coefficient files. Collects group constants in in multi-dimensional matrices according to perturbations, universes, and burnup.
- Plotting routines now use attach to the active plot or generate a new plot figure if `ax` argument not given - #267
- *BranchingReader* can read coefficient files with uncertainties - #270

Warning: The API for the *BranchCollector* may be subject to change through revisions until `0.7.0`

3.8.1 Pending Deprecations

- *saveAsMatlab()* in favor of `serpentTools.io.toMatlab()` with:

```
>>> from serpentTools.io import toMatlab
>>> toMatlab(depR)
```

- Depletion plot routines will no longer accept `timePoints` arguments, instead plotting against all points in time

3.9 0.6.1

- #256 `serpentTools.settings.rc.loadYaml()` uses `safe_load`
- #257 *DepletionReader* now can utilize *saveAsMatlab()* for exporting data to a binary `.mat` file
- #259 Little more clarity into supported readers through documentation and *serpentTools.read()* function

3.10 0.6.0

- #174 - Added parent object `BaseObject` with basic comparison method from which all objects inherit. Comparison method contains upper and lower bounds for values w/o uncertainties, #191
- #196 - Add comparison methods for *ResultsReader* and *HomogUniv* objects
- #228 - Add comparison methods for *DetectorReader* and *Detector* objects
- #236 - Add comparison methods for *DepletionReader* and *DepletedMaterial* objects
- #241 - Fix a bug in the CLI that rendered the ability to generate files with unique random seeds. `python -m serpentTools seed <input> <N>` can now be properly used.
- #249 - Better sparse support for depletion matrix, `depmtx` files with a *DepmtxReader*
- #252 - Better axis and colorbar labeling for *Detector* mesh plots
- #254 - Better plotting of single concentrations with *DepmtxReader*
- #255 - *DepletionReader* can capture material with underscores now!

3.10.1 Deprecations

- `depmtx()` is deprecated in favor of either `readDepmtx()` or the class-based *DepmtxReader*

3.11 0.5.4

- #239 - Update python dependencies to continue use of python 2

3.12 0.5.3

- #221 - Expanded `utils` module to better assist developers
- #227 - Better documentation of our *Command Line Interface*. Better documentation and testing of functions for generating input files with unique random seeds - `serpentTools.seed`
- #229 - *`serpentTools.SensitivityReader.plot()`* now respects the option to not set x nor y labels.
- #231 - *`ResultsReader`* objects can now read files that do not contain group constant data. The setting `results-expectGcu` should be used to inform the reader that no group constant data is anticipated

> **Note:** This setting was removed in *0.8.0* and in #324

3.13 0.5.2

- #198 - Import test and example files using `serpentTools.data`. Load example readers with *`serpentTools.data.readDataFile()`*
- #199 - Support for structured or unstructured matrix plotting with *`serpentTools.plot.cartMeshPlot()`*
- #201 - Support for plotting hexagonal meshes with `serpentTools.objects.HexagonalDetector.hexPlot()`
- #204 - Access *`Detector`* objects directly from *`DetectorReader`* with `reader[detName]`
- #205 - Access materials from *`DepletionReader`* and *`serpentTools.samplers.DepletionSampler`* using key-like indexing, e.g. `reader[matName] == reader.material[matName]`
- #213 - Better default x-axis labels for simple Detector plots

3.13.1 API Changes

- #194 - Some settings in *`serpentTools.ResultsReader.metadata`* are now stored as `int` or `float`, depending upon their nature. Many of these settings refer to flags of settings used by `SERPENT`

3.14 0.5.1

- #180 - Add capability to pass isotope `zzaaai` for `getValues()` and associated plot routines
- #187 - Import all readers and samplers from the main package:

```
>>> from serpentTools import ResultsReader
>>> from serpentTools import DetectorSampler
```

- #189 - Support for reading Detectors with hexagonal, cylindrical, and spherical meshes.

3.14.1 API Changes

- `zzaaai` data is stored on *`zai`* as a list of integers, not strings

3.15 0.5.0

- #131 Updated variable groups between `2.1.29` and `2.1.30` - include poison cross section, kinetic parameters, six factor formula (2.1.30 exclusive), and minor differences
- #141 - Setting *xs.reshapeScatter* can be used to reshape scatter matrices on *`HomogUniv`* objects to square matrices
- #145 - *`hasData()`* added to check if *`HomogUniv`* objects have any data stored on them
- #146 - *`HomogUniv`* object stores group structure on the object. New dictionaries for storing group constant data that is not `INF` nor `B1` - *`gc`* and *`gcUnc`*
- #130 Added the ability to read results file
- #149 - Add the ability to read sensitivity files
- #161 - Add the `utils` module
- #165 - Add the *`serpentTools.objects.HomogUniv.plot()`* method

3.15.1 API Changes

- #146 removed `metadata` dictionaries on *`HomogUniv`* objects.

3.15.2 Deprecation

- Variable group `xs-yields` is removed. Use `poisons` instead
- Branches of a single name are only be accessible through `branches['nom']`, not `branches[('nom'),]` as per #114

3.16 0.4.0

- #95 Add `xsplot` file reader - *`XSPlotReader`*
- #121 Samplers will raise more warnings/errors if no files are loaded from `*` wildcards
- #122 Better Detector labeling
- #135 Added instructions for better converting Jupyter notebooks to `.rst` files. Plotting guidelines

3.17 0.3.1

- #118 - Support for `SERPENT` 2.1.30
- #119 - SampledDepletedMaterial now respects the value of *xUnits* - #120
- #114 - Standalone branches in the coefficient files are stored and accessed using a single string, rather than a single-entry tuple `branches['myBranch']` vs. `branches[('myBranch', )]`

3.18 0.3.0

- #109 - Capability to read history files
- #107 - *`DepletionReader`* can now plot data for some or all materials

3.19 0.2.2

- #82 - Command line interface and some sub-commands
- #88 - Pre- and post-check methods for readers
- #93 - Detector and Depletion Samplers
- #96 - Better mesh plotting for Detector
- #99 - Negative universe burnup with branching reader - #100
- `serpentTools.objects.Detector.indexes` are now zero-indexed
- The PDF manual is no longer tracked in this repository

CHAPTER

FOUR

EXAMPLES

4.1 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 *`serpentTools.settings.rc`* class.

All the settings are given in *Default Settings*, showing their default values and possible options.

4.1.1 Basic Usage

```
>>> import serpentTools
>>> from serpentTools.settings import rc
```

The *`serpentTools.settings.rc`* object can be used like a dictionary, polling all possible settings with the `.keys` method:

```
>>> rc.keys()
dict_keys(['depletion.processTotal', 'verbosity', 'xs.getInfXS',
'branching.intVariables', 'serpentVersion',
'sampler.raiseErrors', 'xs.getB1XS', 'xs.variableGroups', 'xs.variableExtras',
'depletion.materialVariables', 'depletion.metadataKeys', 'sampler.freeAll',
'sampler.allExist', 'depletion.materials', 'sampler.skipPrecheck',
'xs.reshapeScatter', 'branching.floatVariables', 'detector.names'])
```

Settings such as *depletion.materials* 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. Updating a setting is similar to setting a value in a dictionary

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

The *`serpentTools.settings.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:
```

(continues on next page)

(continued from previous page)

```
...     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,
2.1.30'
```

The *serpentTools.settings.rc* object can also be used inside a context manager to revert changes.

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

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 described in *Variable Groups* 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', 'BETA_EFF',
'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', 'LAMBDA', '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.variableExtras* and *xs.variableGroups* 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

The *serpentTools.settings.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
```

However, the loader can also expand a nested dictionary structure, as:

```
branching:
  floatVariables: [Fhi, Blo]
depletion:
  materials: [fuel*]
  materialVariables:
    [ADENS, MDENS, VOLUME]
```

```
>>> %cat myConfig.yaml
xs.getInfXS: False
xs.getB1XS: True
xs.variableGroups: [gc-meta, kinetics,
xs]
branching:
  floatVariables: [Fhi, Blo]
depletion:
  materials: [fuel*]
  metadataKeys: [NAMES, BU]
materialVariables:
    [ADENS, MDENS, VOLUME]
serpentVersion: 2.1.29
>>> myConf = 'myConfig.yaml'
>>> rc.loadYaml(myConf)
INFO    : serpentTools: Done
>>> rc['xs.getInfXS']
False
```

4.2 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

4.2.1 Basic Operation

Note: The preferred way to read your own output files is with the *serpentTools.read()* function. The `serpentTools.readDataFile()` function is used here to make it easier to reproduce the examples

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'
>>> r0 = serpentTools.readDataFile(branchFile)
```

The branches are stored in custom dictionary-like *BranchContainer* objects in the *branches* dictionary

```
>>> r0.branches.keys()
dict_keys([
    ('nom', 'nom'),
    ('B750', 'nom'),
    ('B1000', 'nom'),
    ('nom', 'FT1200'),
    ('B750', 'FT1200'),
    ('B1000', 'FT1200'),
    ('nom', 'FT600'),
    ('B750', 'FT600'),
    ('B1000', 'FT600')
])
```

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 as a dictionary.

```
>>> for key in b0:
...     print(key)
UnivTuple(universe='0', burnup=0.0, step=0, days=None)
UnivTuple(universe='10', burnup=0.0, step=0, days=None)
UnivTuple(universe='20', burnup=0.0, step=0, days=None)
UnivTuple(universe='30', burnup=0.0, step=0, days=None)
UnivTuple(universe='40', burnup=0.0, step=0, days=None)
UnivTuple(universe='0', burnup=1.0, step=1, days=None)
UnivTuple(universe='10', burnup=1.0, step=1, days=None)
UnivTuple(universe='20', burnup=1.0, step=1, days=None)
UnivTuple(universe='30', burnup=1.0, step=1, days=None)
UnivTuple(universe='40', burnup=1.0, step=1, days=None)
UnivTuple(universe='0', burnup=10.0, step=2, days=None)
UnivTuple(universe='10', burnup=10.0, step=2, days=None)
UnivTuple(universe='20', burnup=10.0, step=2, days=None)
UnivTuple(universe='30', burnup=10.0, step=2, days=None)
UnivTuple(universe='40', burnup=10.0, step=2, days=None)
```

The keys here are `UnivTuple` instances indicating the universe ID, and point in the burnup schedule. These universes can be obtained by indexing this dictionary, or by using the *getUniv()* method

```
>>> univ0 = b0["0", 1, 1, None]
>>> print(univ0)
<HomogUniv 0: burnup: 1.000 MWd/kgu, step: 1>
>>> univ0.name, univ0.bu, univ0.step, univ0.day
('0', 1.0, 1, None)
>>> univ1 = b0.getUniv('0', burnup=1)
>>> univ2 = b0.getUniv('0', index=1)
>>> univ0 is univ1 is univ2
True
```

Group constant data is spread out across the following sub-dictionaries:

1. *infExp*: Expected values for infinite medium group constants
2. *infUnc*: Relative uncertainties for infinite medium group constants
3. *b1Exp*: Expected values for leakage-corrected group constants
4. *b1Unc*: Relative uncertainties for leakage-corrected group constants
5. *gc*: Group constant data that does not match the `INF` nor `B1` scheme
6. *gcUnc*: Relative uncertainties for data in *gc*

For this problem, only expected values for infinite and critical spectrum (b1) group constants are returned, so only the `infExp` and `b1Exp` dictionaries contain data

```
>>> univ0.infExp
{'infDiffcoef': array([ 1.83961 ,  0.682022]),
 'infFiss': array([ 0.00271604,  0.059773  ]),
 'infS0': array([ 0.298689  ,  0.00197521,  0.00284247,  0.470054  ]),
```

(continues on next page)

(continued from previous page)

```
 '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 ]),
 '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.gc
{}
>>> univ0.gcUnc
{}
```

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

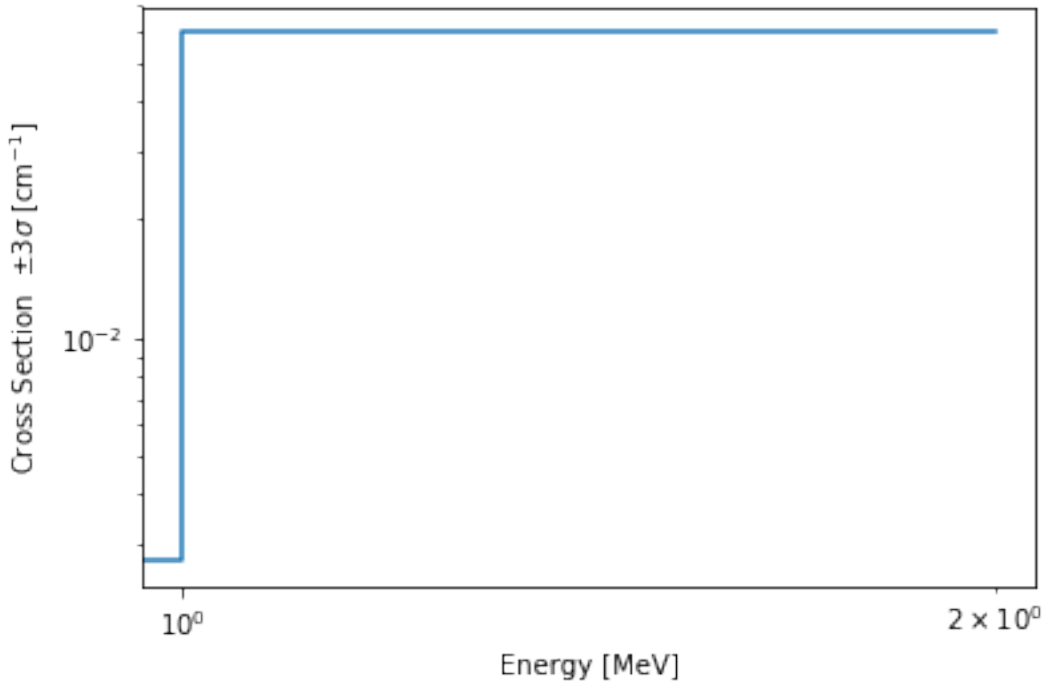
```
>>> univ0.get('infFiss')
array([ 0.00271604,  0.059773  ])

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

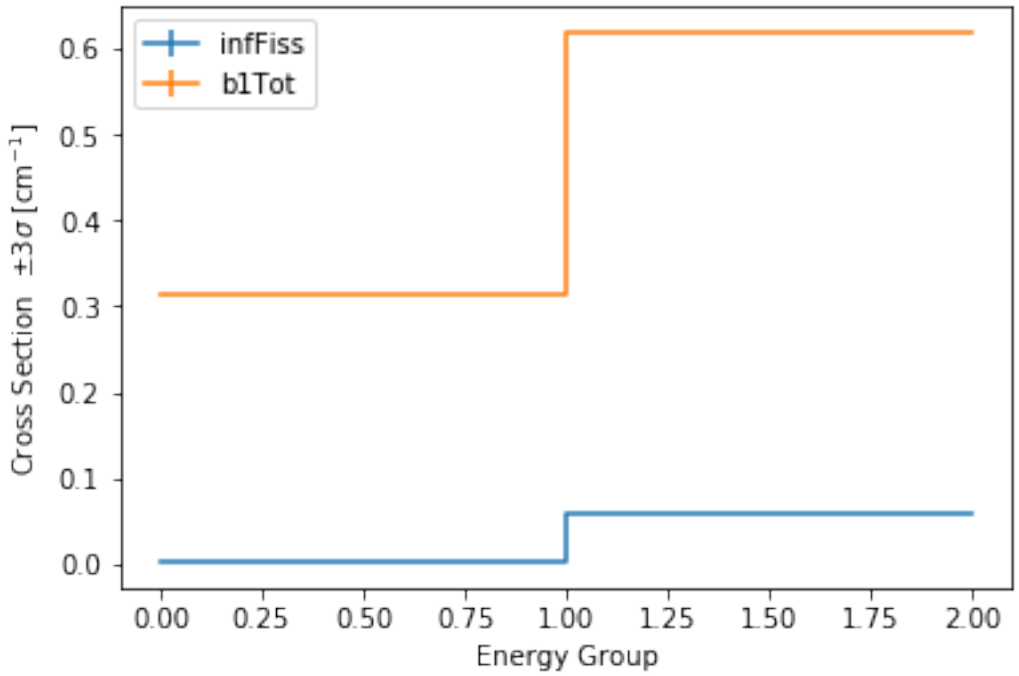
4.2.2 Plotting Universe Data

HomogUniv objects are capable of plotting homogenized data using the *plot()* method. This method is tuned to plot group constants, such as cross sections, for a known group structure. This is reflected in the default axis scaling, but can be adjusted on a per case basis. If the group structure is not known, then the data is plotted simply against bin-index.

```
>>> univ0.plot('infFiss');
```




```
>>> univ0.plot(['infFiss', 'b1Tot'], loglog=False);
```



The `ResultsReader` example has a more thorough example of this `plot()` method, including formatting the line labels - *Plotting universes*.

4.2.3 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', 'FT1200') <BranchContainer for nom, FT1200 from demo.coe>
('B1000', 'FT1200') <BranchContainer for B1000, FT1200 from demo.coe>
('B750', 'FT600') <BranchContainer for B750, FT600 from demo.coe>
('nom', 'nom') <BranchContainer for nom, nom from demo.coe>
('B750', 'FT1200') <BranchContainer for B750, FT1200 from demo.coe>
('B1000', 'FT600') <BranchContainer for B1000, FT600 from demo.coe>
('nom', 'FT600') <BranchContainer for nom, FT600 from demo.coe>
('B1000', 'nom') <BranchContainer for B1000, nom from demo.coe>
('B750', 'nom') <BranchContainer for B750, nom from demo.coe>
```

4.2.4 User Control

The SERPENT set coefpara card already restricts the data present in the coefficient file to user control, and the *BranchingReader* includes similar control.

- *branching.floatVariables*
- *branching.intVariables*
- *xs.getB1XS*
- *xs.getInfXS*
- *xs.reshapeScatter*
- *xs.variableExtras*
- *xs.variableGroups*

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 *branching.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.variableExtras* and *xs.variableGroups* controls what data is stored on the *HomogUniv* objects. By default, all variables present in the coefficient file are stored.

```
>>> from serpentTools.settings import rc
>>> rc['branching.floatVariables'] = ['BOR']
>>> rc['branching.intVariables'] = ['TFU']
>>> rc['xs.getB1XS'] = False
>>> rc['xs.variableExtras'] = ['INF_TOT', 'INF_SCATT0']
>>> r1 = serpentTools.readDataFile(branchFile)
>>> b1 = r1.branches['B1000', 'FT600']
>>> b1.stateData
{'BOR': 1000.0,
 'DATE': '17/12/19',
 'TFU': 600,
 'TIME': '09:48:54',
 'VERSION': '2.1.29'}
```

(continues on next page)

(continued from previous page)

```
>>> 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
{}
```

4.2.5 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.

4.3 Detector Reader

4.3.1 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.

Note: The preferred way to read your own output files is with the *serpentTools.read()* function. The `serpentTools.readDataFile()` function is used here to make it easier to reproduce the examples

---

```
>>> from matplotlib import pyplot
>>> import serpentTools
>>> pinFile = 'fuelPin_det0.m'
>>> bwrFile = 'bwr_det0.m'
>>> pin = serpentTools.readDataFile(pinFile)
>>> bwr = serpentTools.readDataFile(bwrFile)
>>> print(pin.detectors)
{'nodeFlx': <serpentTools.Detector object at 0x7f6df2162b70>}
>>> print(bwr.detectors)
{'xymesh': <serpentTools.Detector object at 0x7f6df2162a90>,
 'spectrum': <serpentTools.Detector object at 0x7f6df2162b00>}
```

These detectors were defined for a single fuel pin with 16 axial layers and a separate BWR assembly, with a description of the detectors provided in below:

Name	Description
`nodeFlx`	One-group flux tallied in each axial layer
`spectrum`	CSEWG 239 group stucture for flux and U-235 fission cross section
`xymesh`	Two-group flux for a 20x20 xy grid

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

```
>>> nodeFlx = pin.detectors['nodeFlx']
>>> print(nodeFlx.bins.shape)
(16, 12)
>>> nodeFlx.bins[:3,:].T
array([[1.00000e+00, 2.00000e+00, 3.00000e+00],
       [1.00000e+00, 1.00000e+00, 1.00000e+00],
       [1.00000e+00, 2.00000e+00, 3.00000e+00],
       [1.00000e+00, 1.00000e+00, 1.00000e+00],
       [1.00000e+00, 1.00000e+00, 1.00000e+00],
       [1.00000e+00, 1.00000e+00, 1.00000e+00],
       [1.00000e+00, 1.00000e+00, 1.00000e+00],
       [1.00000e+00, 1.00000e+00, 1.00000e+00],
       [1.00000e+00, 1.00000e+00, 1.00000e+00],
       [1.00000e+00, 1.00000e+00, 1.00000e+00],
       [2.34759e-02, 5.75300e-02, 8.47000e-02],
       [4.53000e-03, 3.38000e-03, 2.95000e-03]])
```

Here, only three columns, shown as rows for readability, are changing:

- column 0: universe column
- column 10: tally column
- column 11: errors

Detectors can also be obtained by indexing into the *DetectorReader*, as

```
>>> nf = pin['nodeFlx']
>>> assert nf is nodeFlx
```

Tally data is reshaped corresponding to the bin information provided by Serpent. The tally and errors columns are recast into multi-dimensional arrays where each dimension is some unique bin type like energy or spatial bin index. For this case, since the only variable bin quantity is that of the universe, the `tallies` and `errors` attributes will 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])
```

Note: Python and numpy arrays are zero-indexed, meaning the first item is accessed with `array[0]`, rather than `array[1]`.

Bin information is retained through the *indexes* attribute. Each entry indicates what bin type is changing along that dimension of `tallies` and `errors`. Here, `universe` is the first item and indicates that the first dimension of `tallies` and `errors` corresponds to a changing universe bin.

```
>>> nodeFlx.indexes
("universe", )
```

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 = bwr.detectors['spectrum']
>>> print(spectrum.grids['E'][:5])
[[1.00002e-11 4.13994e-07 2.07002e-07]
 [4.13994e-07 5.31579e-07 4.72786e-07]
 [5.31579e-07 6.25062e-07 5.78320e-07]
 [6.25062e-07 6.82560e-07 6.53811e-07]
 [6.82560e-07 8.33681e-07 7.58121e-07]]
```

4.3.2 Multi-dimensional Detectors

The *Detector* objects are capable of reshaping the detector data into an 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 = bwr.detectors['xymesh']
>>> xy.indexes
('energy', 'ymesh', 'xmesh')
```

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

```
>>> print(xy.bins.shape)
(800, 12)
>>> print(xy.tallies.shape)
(2, 20, 20)
>>> print(xy.bins[:5, 10])
[8.19312e+17 7.18519e+17 6.90079e+17 6.22241e+17 5.97257e+17]
>>> print(xy.tallies[0, 0, :5])
[8.19312e+17 7.18519e+17 6.90079e+17 6.22241e+17 5.97257e+17]
```

Slicing

As the detectors produced by SERPENT can contain multiple bin types, 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 fission reaction in the `spectrum` detector, you would instruct the *slice()* method to use column 1 along the axis that corresponds to the reaction bin, as the fission reaction corresponded to reaction tally 2 in the original matrix. Since python and *numpy* arrays are zero indexed, the second reaction tally is stored in column 1.

```
>>> spectrum.slice({'reaction': 1})[:20]
array([3.66341e+22, 6.53587e+20, 3.01655e+20, 1.51335e+20, 3.14546e+20,
       7.45742e+19, 4.73387e+20, 2.82554e+20, 9.89379e+19, 9.49670e+19,
       8.98272e+19, 2.04606e+20, 3.58272e+19, 1.44708e+20, 7.25499e+19,
       6.31722e+20, 2.89445e+20, 2.15484e+20, 3.59303e+20, 3.15000e+20])
```

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

```
>>> spectrum.slice({'reaction': 1}, 'errors')[:20]
array([0.00692, 0.01136, 0.01679, 0.02262, 0.01537, 0.02915, 0.01456,
       0.01597, 0.01439, 0.01461, 0.01634, 0.01336, 0.01549, 0.01958,
       0.02165, 0.0192 , 0.02048, 0.01715, 0.02055, 0.0153 ])
```

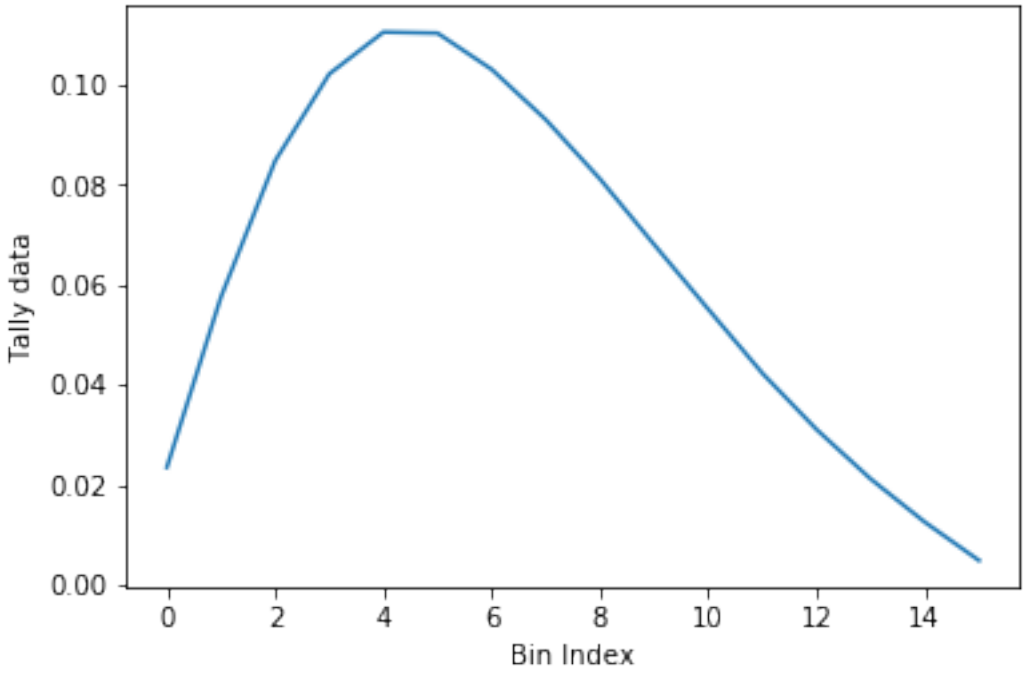
4.3.3 Plotting Routines

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. Below are keyword arguments that can be used to format the plots.

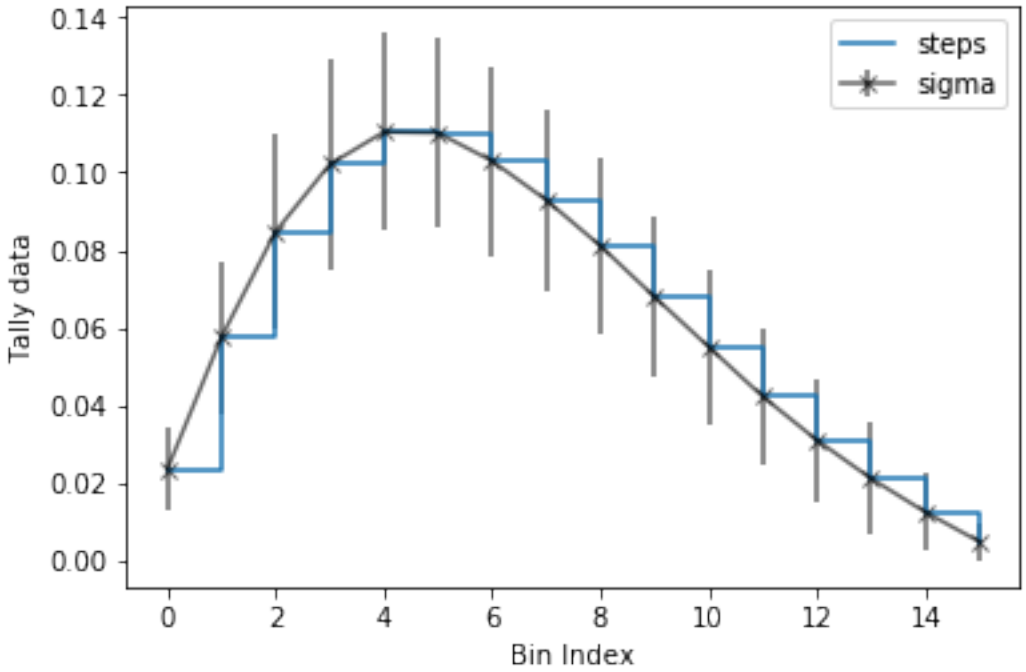
option	description
what	what data to plot
ax	preprepared figure on which to add this plot
xdim	quantity from indexes to use as x-axis
sigma	confidence interval to place on errors - 1d
steps	draw tally values as constant inside bin - 1d
xlabel	label to apply to x-axis
ylabel	label to apply to y-axis
loglog	use a log scalling on both of the axes
logx	use a log scaling on the x-axis
logy	use a log scaling on the y-axis
legend	place a legend on the figure
ncol	number of columns to apply to the legend

The plot routine also accepts various options, which can be found in the matplotlib.pyplot.plot documentation

```
>>> nodeFlx.plot()
```

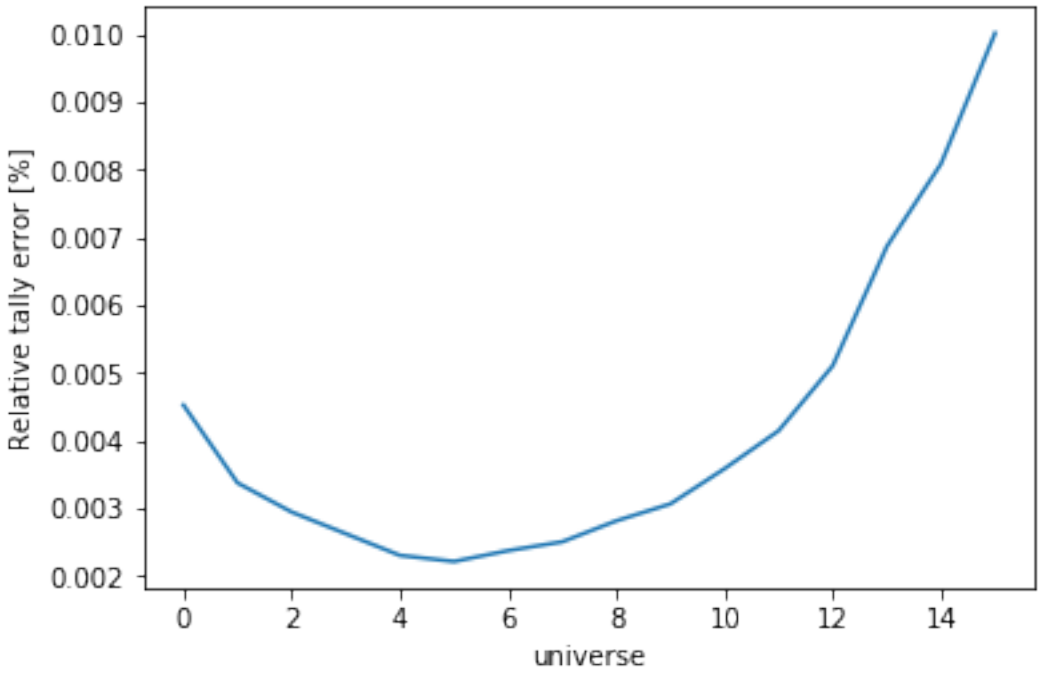



```
>>> ax = nodeFlx.plot(steps=True, label='steps')
>>> ax = nodeFlx.plot(sigma=100, ax=ax, c='k', alpha=0.6,
...                   marker='x', label='sigma')
```



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`* as the `xdim` argument sets the x-axis to be that specific index.

```
>>> nodeFlx.plot(xdim='universe', what='errors',
...              ylabel='Relative tally error [%]')
```



Mesh Plots

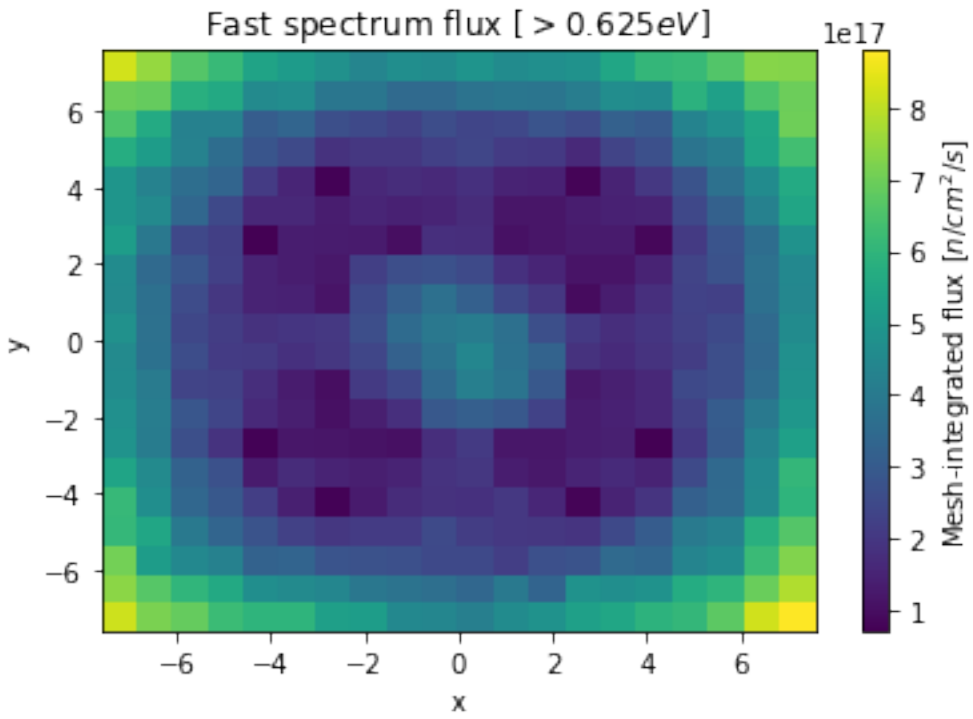
For data with dimensionality greater than one, the *meshPlot()* method can be used to plot some 2D slice of the data on a Cartesian grid. Passing a dictionary as the `fixed` argument restricts the tally data down to two dimensions. The X and Y axis can be quantities from *grids* or *indexes*. If the quantity to be used for an axis is in the *grids* dictionary, then the appropriate spatial or energetic grid from the detector file will be used. Otherwise, the axis will reflect changes in a specific bin type. The following keyword arguments can be used in conjunction with the above options to format the mesh plots.

Option	Action
`cmap`	Colormap to apply to the figure
`cbarLabel`	Label to apply to the colorbar
`logColor`	If true, use a logarithmic scale for the colormap
`normalizer`	Apply a custom non-linear normalizer to the colormap

The `cmap` argument must be something that `matplotlib` can understand as a valid colormap. This can be a string of any of the colormaps supported by *matplotlib*.

Since the `xymesh` detector is three dimensions, (energy, x, and y), we must pick an energy group to plot.

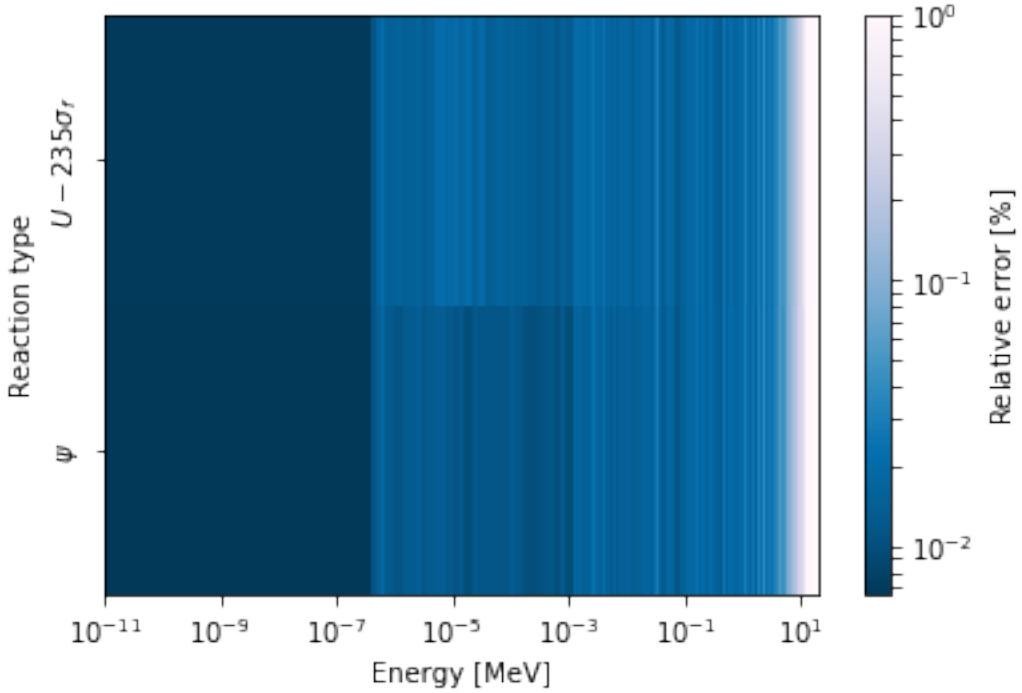
```
>>> xy.meshPlot('x', 'y', fixed={'energy': 0},
...             cbarLabel='Mesh-integrated flux $[n/cm^2/s]$',
...             title="Fast spectrum flux $[>0.625 eV]$");
```



The *meshPlot()* also supports a range of labeling and plot options. Here, we attempt to plot the flux and U-235 fission reaction rate errors as a function of energy, with the two reaction rates separated on the y-axis. Passing `logColor=True` applies a logarithmic color scale to all the positive data. Data that is zero is not shown, and errors will be raised if the data contain negative quantities.

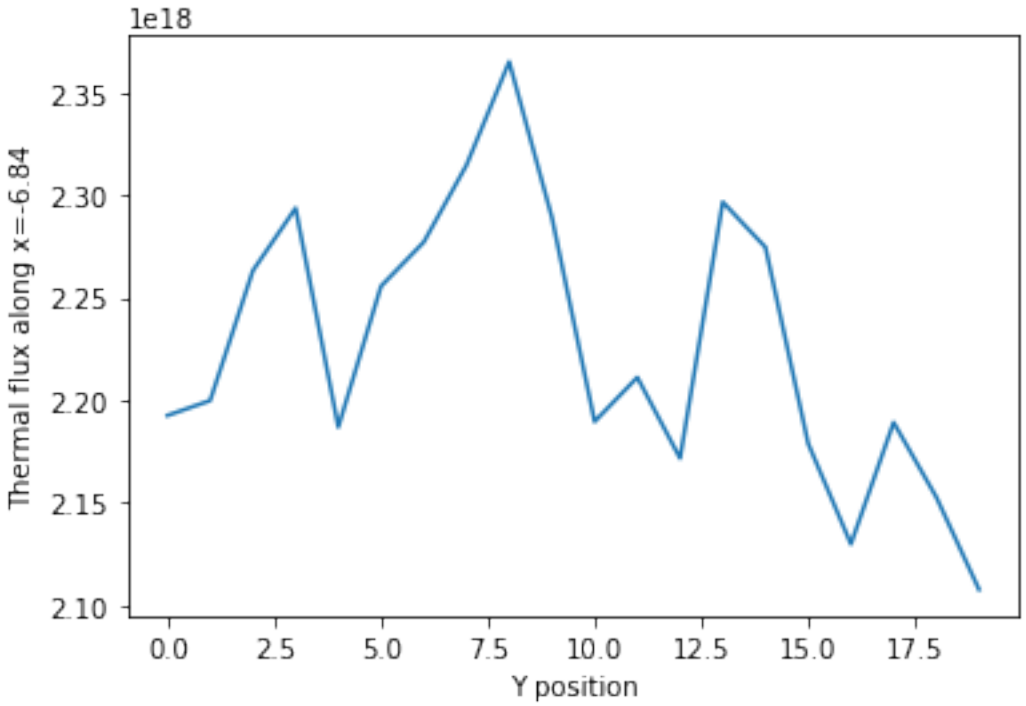
Here we also apply custom y-tick labels to reflect the reaction that is being plotted.

```
>>> ax = spectrum.meshPlot('e', 'reaction', what='errors',
...                        ylabel='Reaction type', cmap='PuBu_r',
...                        cbarLabel="Relative error $[\%]$",
...                        xlabel='Energy [MeV]', logColor=True,
...                        logx=True);
>>> ax.set_yticks([0.5, 1.5]);
>>> ax.set_yticklabels([r'$\psi$', r'$U-235 \sigma_f$'], rotation=90,
>>>                    verticalalignment='center');
```



Using the `slicing` arguments allows access to the 1D plot methods from before

```
>>> xy.plot(fixed={'energy': 1, 'xmesh': 1},
...         xlabel='Y position',
...         ylabel='Thermal flux along x={}'
...         .format(xy.grids['X'][1, 0]));
```



Spectrum Plots

The *Detector* objects are also capable of energy spectrum plots, if an associated energy grid is given. The `normalize` option will normalize the data per unit lethargy. This plot takes some additional assumptions with the scaling and labeling, but all the same controls as the above line plots.

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

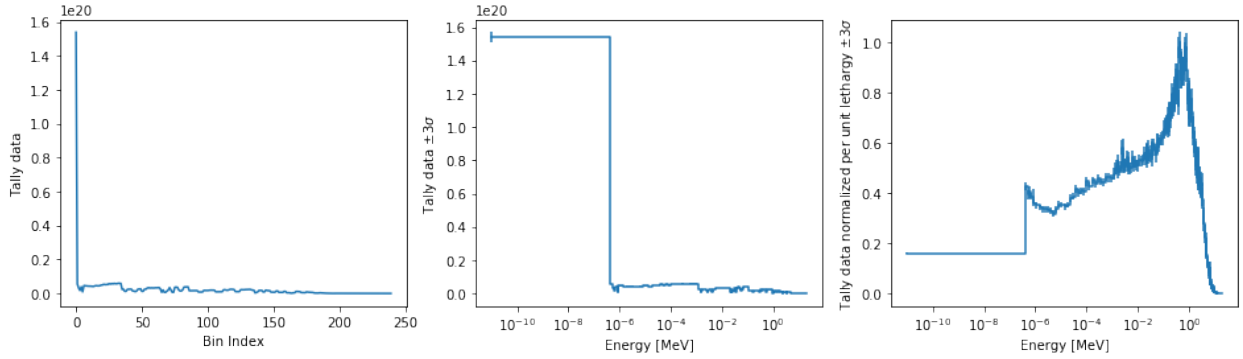
Option	Default	Description
`normalize`	`True`	Normalize tallies per unit lethargy
`fixed`	`None`	Dictionary that controls matrix reduction
`sigma`	3	Level of confidence for statistical errors
`xscale`	`'log'`	Set the x scale to be log or linear
`yscale`	`'linear'`	Set the y scale to be log or linear

The figure below demonstrates the default options and control in this *spectrumPlot()* routine by

1. Using the less than helpful plot routine with no formatting
2. Using *spectrumPlot()* without normalization to show default labels and scaling
3. Using *spectrumPlot()* with normalization

Since our detector has energy bins and reaction bins, we need to reduce down to one-dimension with the `fixed` command.

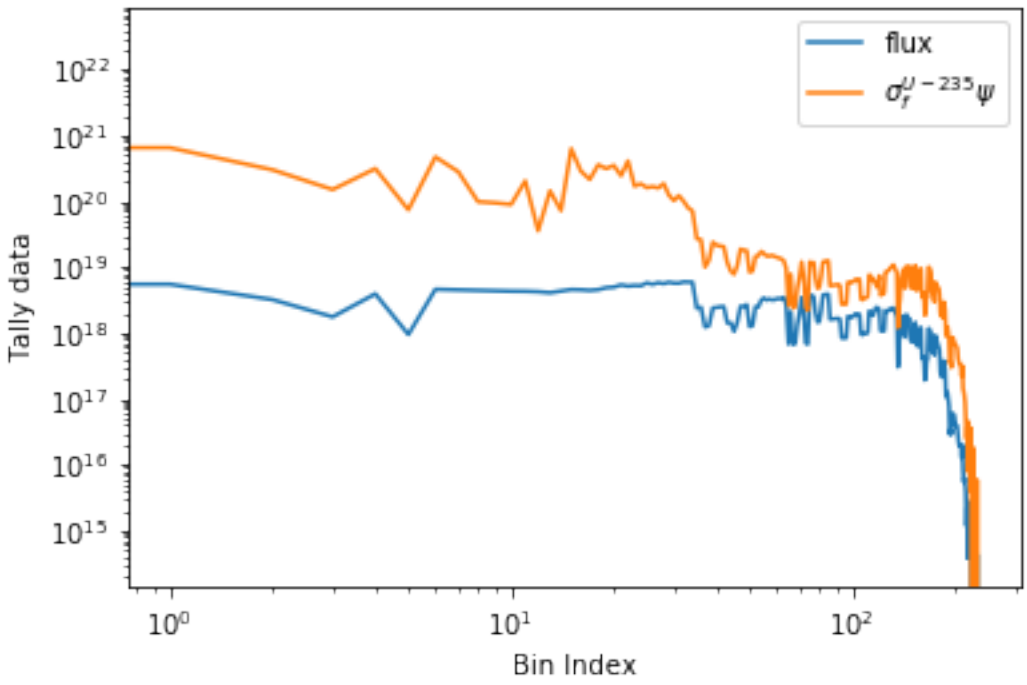
```
>>> fig, axes = pyplot.subplots(1, 3, figsize=(16, 4))
>>> fix = {'reaction': 0}
>>> spectrum.plot(fixed=fix, ax=axes[0]);
>>> spectrum.spectrumPlot(fixed=fix, ax=axes[1], normalize=False);
>>> spectrum.spectrumPlot(fixed=fix, ax=axes[2]);
```



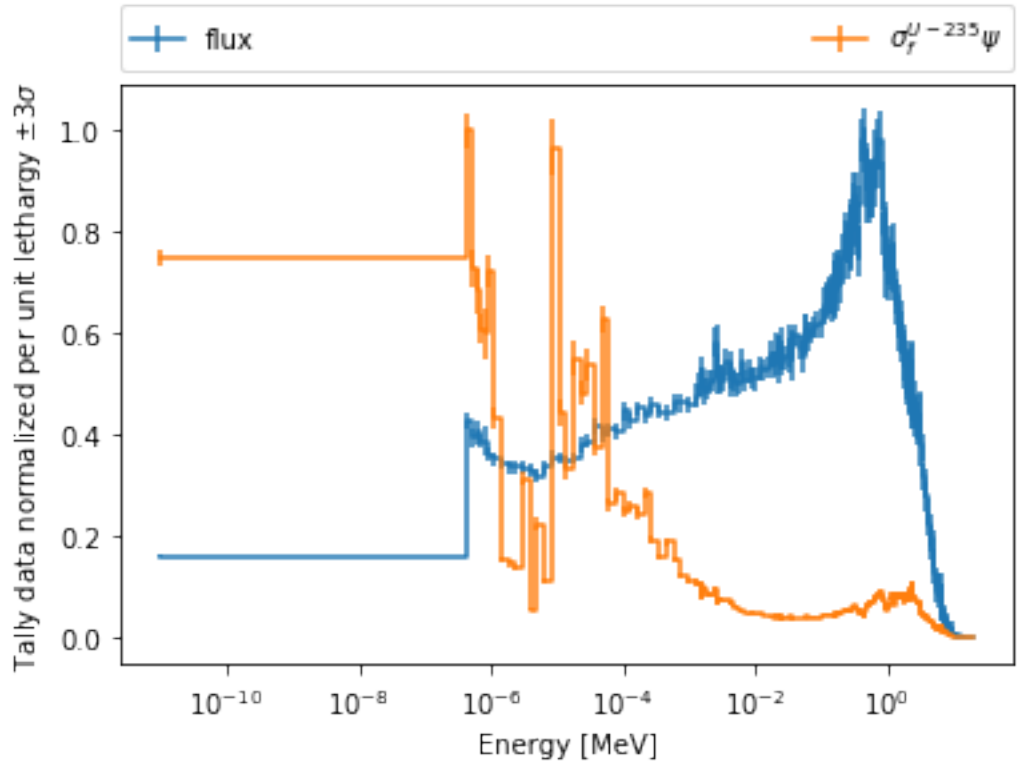
Multiple line plots

Plots can be made against multiple bins, such as spectrum in different materials or reactions, with the *plot()* and *spectrumPlot()* methods. Below is the flux spectrum and spectrum of the U-235 fission reaction rate from the same detector. The `labels` argument is what is used to label each individual plot in the order of the bin index.

```
>>> labels = (
...     'flux',
...     r'$\sigma_f^{U-235}\psi$')  # render as mathtype
>>> spectrum.plot(labels=labels, loglog=True);
```




```
>>> spectrum.spectrumPlot(labels=labels, legend='above', ncol=2);
```



4.3.4 Hexagonal Detectors

SERPENT allows the creation of hexagonal detectors with the `dh` card, like:

```
det hex2 2 0.0 0.0 1 5 5 0.0 0.0 1
det hex3 3 0.0 0.0 1 5 5 0.0 0.0 1
```

which would create two hexagonal detectors with different orientations. Type 2 detectors have two faces perpendicular to the x-axis, while type 3 detectors have faces perpendicular to the y-axis. For more information, see the dh card from SERPENT wiki.

`serpentTools` is capable of storing data tallies and grid structures from hexagonal detectors in *`HexagonalDetector`* objects.

```
>>> hexFile = 'hexplot_det0.m'
>>> hexR = serpentTools.readDataFile(hexFile)
>>> hexR.detectors
{'hex2': <serpentTools.HexagonalDetector at 0x7f1ad03d5da0>,
'hex3': <serpentTools.HexagonalDetector at 0x7f1ad03d5c88>}
```

Here, two *`HexagonalDetector`* objects are produced, with similar *`tallies`* and slicing methods as demonstrated above.

```
>>> hex2 = hexR.detectors['hex2']
>>> hex2.tallies
array([[0.185251, 0.184889, 0.189381, 0.184545, 0.195442],
       [0.181565, 0.186038, 0.193088, 0.195448, 0.195652],
       [0.1856  , 0.190278, 0.192013, 0.193353, 0.184309],
       [0.186249, 0.191939, 0.192513, 0.194196, 0.186953],
       [0.198196, 0.198623, 0.195612, 0.174804, 0.178053]])
>>> hex2.grids
{'COORD': array([[-3.       , -1.732051 ],
        [-2.5      , -0.8660254],
        [-2.       ,  0.       ],
        [-1.5      ,  0.8660254],
        [-1.       ,  1.732051 ],
        [-2.       , -1.732051 ],
        [-1.5      , -0.8660254],
        [-1.       ,  0.       ],
        [-0.5      ,  0.8660254],
        [ 0.       ,  1.732051 ],
        [-1.       , -1.732051 ],
        [-0.5      , -0.8660254],
        [ 0.       ,  0.       ],
        [ 0.5      ,  0.8660254],
        [ 1.       ,  1.732051 ],
        [ 0.       , -1.732051 ],
        [ 0.5      , -0.8660254],
        [ 1.       ,  0.       ],
        [ 1.5      ,  0.8660254],
        [ 2.       ,  1.732051 ],
        [ 1.       , -1.732051 ],
        [ 1.5      , -0.8660254],
        [ 2.       ,  0.       ],
        [ 2.5      ,  0.8660254],
        [ 3.       ,  1.732051 ]]),
 'Z': array([[0., 0., 0.]])}
```

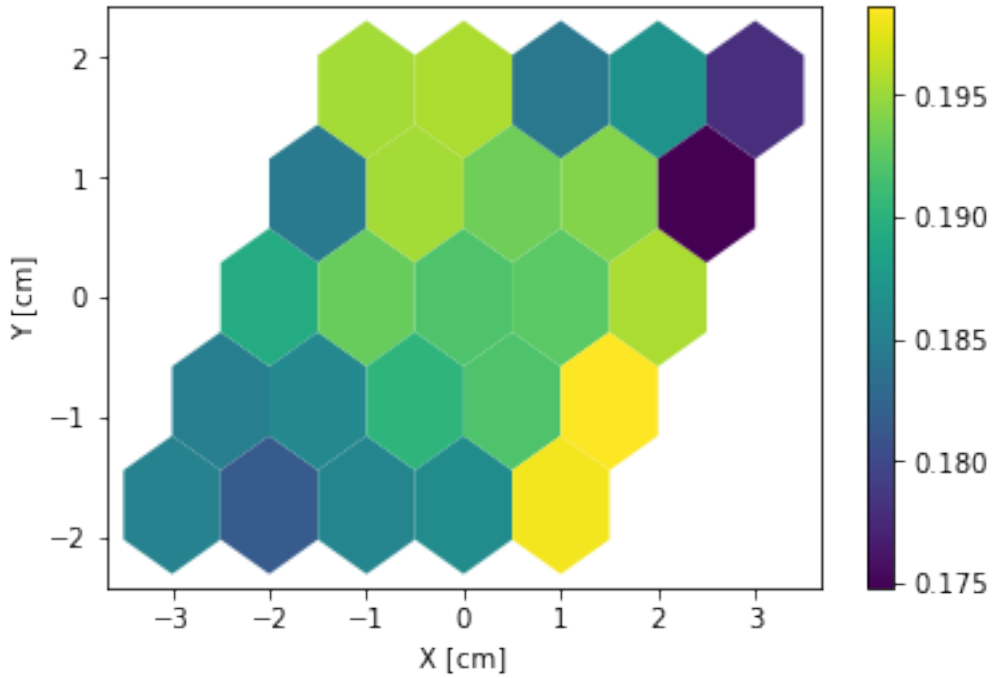
(continues on next page)

(continued from previous page)

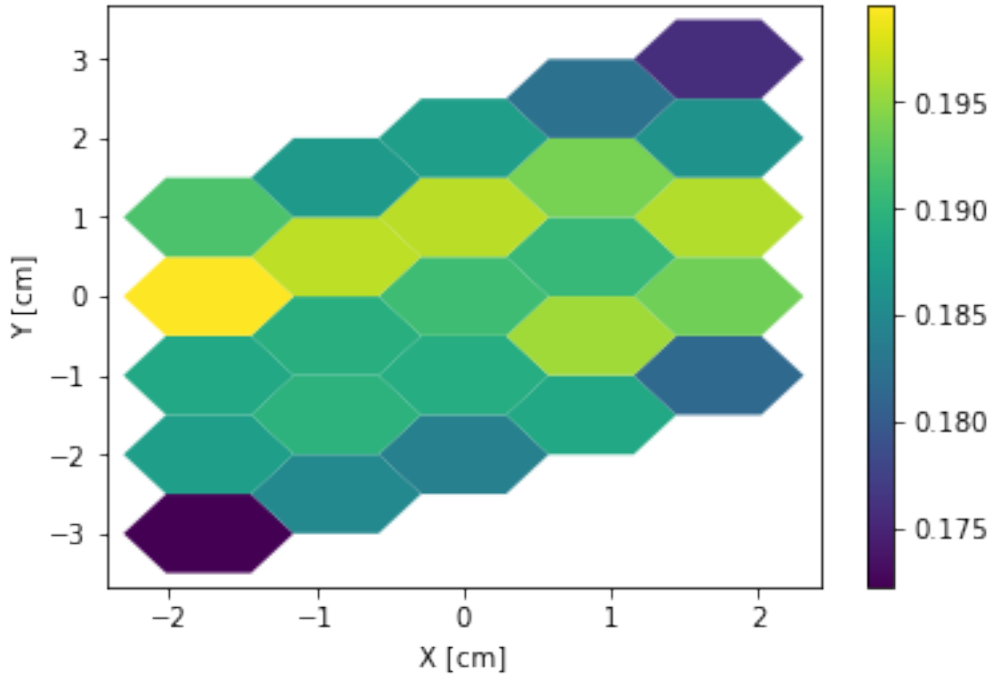
```
>>> hex2.indexes
('ycoord', 'xcoord')
```

Creating hexagonal mesh plots with these objects requires setting the *pitch* and *hexType* attributes.

```
>>> hex2.pitch = 1
>>> hex2.hexType = 2
>>> hex2.hexPlot();
```




```
>>> hex3 = hexR.detectors['hex3']
>>> hex3.pitch = 1
>>> hex3.hexType = 3
>>> hex3.hexPlot();
```



4.3.5 Limitations

`serpentTools` does support reading detector files with hexagonal, cylindrical, and spherical mesh structures. However, creating 2D mesh plots with cylindrical and spherical detectors, and utilizing their mesh structure, is not fully supported. #169 is currently tracking progress for cylindrical plotting.

4.3.6 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.

4.3.7 References

1. matplotlib plot
2. Custom colormap normalization
3. matplotlib 2.0 colormaps

4.4 Depletion Reader

4.4.1 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 serpentTools
>>> from serpentTools.settings import rc
```

Note: The preferred way to read your own output files is with the *read()* function. The *readDataFile()* function is used here to make it easier to reproduce the examples

```
>>> depFile = 'demo_dep.m'
>>> dep = serpentTools.readDataFile(depFile)
```

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
{'fuel0': <serpentTools.objects.materials.DepletedMaterial at 0x7f8f8dccde80>,
 'bglass0': <serpentTools.objects.materials.DepletedMaterial at 0x7f8f8f42f518>,
 'total': <serpentTools.objects.materials.DepletedMaterial at 0x7f8f8dce7940>}
```

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

```
>>> dep.metadata.keys()
dict_keys(['zai', 'names', 'burnup', 'days'])
>>> dep.metadata['burnup']
array([0.  , 0.02, 0.04, 0.06, 0.08, 0.1 , 0.12, 0.14, 0.16, 0.18, 0.2 ,
       0.22, 0.24, 0.26, 0.28, 0.3 , 0.32, 0.34, 0.36, 0.38, 0.4 , 0.42,
       0.44, 0.46, 0.48, 0.5 , 0.52, 0.54, 0.56, 0.58, 0.6 , 0.62, 0.64,
       0.66, 0.68, 0.7 , 0.72, 0.74, 0.76, 0.78, 0.8 , 0.82, 0.84, 0.86,
       0.88, 0.9 , 0.92, 0.94, 0.96, 0.98, 1.  , 1.02, 1.04, 1.06, 1.08,
       1.1 , 1.12, 1.14, 1.16, 1.18, 1.2 , 1.22, 1.24, 1.26, 1.28, 1.3 ,
       1.32, 1.34, 1.36, 1.38, 1.4 , 1.42])
>>> dep.metadata['names']
['Xe135', 'I135', 'U234', 'U235', 'U236', 'U238',
 'Pu238', 'Pu239', 'Pu240', 'Pu241', 'Pu242', 'Np237',
 'Am241', 'Am243', 'Cm243', 'Cm244', 'Cm245', 'Cs133',
 'Nd143', 'Sm147', 'Sm149', 'Sm150', 'Sm151', 'Sm152',
 'Eu153', 'Gd155', 'Mo95', 'Tc99', 'Ru101', 'Rh103',
 'Ag109', 'Cd113', 'lost', 'total']
```

4.4.2 Depleted Material Objects

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']
>>> print(fuel.burnup)
[0.         0.00702676 0.0144405  0.0218803  0.0297245  0.0370823
 0.0447201  0.0513465  0.0590267  0.0671439  0.073392   0.0802637
 0.0887954  0.0974604  0.104807   0.111528   0.119688   0.128006
 0.135704   0.143491   0.150545   0.157608   0.165391   0.172872
 0.180039   0.188011   0.195215   0.202291   0.20963    0.216895
 0.224651   0.232139   0.23904    0.246076   0.25422    0.262011
 0.269681   0.276981   0.284588   0.291835   0.299661   0.30727
 0.314781   0.322364   0.329404   0.336926   0.34438    0.352246
 0.360913   0.367336   0.37415    0.381556   0.388951   0.396286
 0.404159   0.412113   0.419194   0.426587   0.43425    0.442316
 0.449562   0.456538   0.465128   0.472592   0.479882   0.487348
 0.494634   0.502167   0.508326   0.515086   0.522826   0.530643  ]
>>> print(fuel.days is dep.metadata['days'])
True
```

Materials can also be obtained by indexing directly into the reader, with

```
>>> newF = dep['fuel0']
>>> assert newF is fuel
```

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(['volume', 'burnup', 'adens', 'mdens', 'a', 'h', 'sf', 'gsrc', 'ingTox',
↪'inhTox'])
>>> print(fuel.adens)
[[0.00000e+00 2.43591e-09 4.03796e-09 ... 4.70133e-09 4.70023e-09
  4.88855e-09]
 [0.00000e+00 6.06880e-09 8.11783e-09 ... 8.05991e-09 8.96359e-09
  9.28554e-09]
 [4.48538e-06 4.48486e-06 4.48432e-06 ... 4.44726e-06 4.44668e-06
  4.44611e-06]
 ...
 [0.00000e+00 3.03589e-11 7.38022e-11 ... 1.62829e-09 1.63566e-09
  1.64477e-09]
 [0.00000e+00 1.15541e-14 2.38378e-14 ... 8.60736e-13 8.73669e-13
  8.86782e-13]
 [6.88332e-02 6.88334e-02 6.88336e-02 ... 6.88455e-02 6.88457e-02
  6.88459e-02]]
>>> print(fuel.adens is fuel.data['adens'])
True
```

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,)
```

(continues on next page)

(continued from previous page)

```
>>> len(fuel.names)
34
```

4.4.3 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. `getValues()` requires two arguments for the units of time requested, e.g. `days` or `burnup`, and the name of the data requested. This second value must be a key in the *data* dictionary.

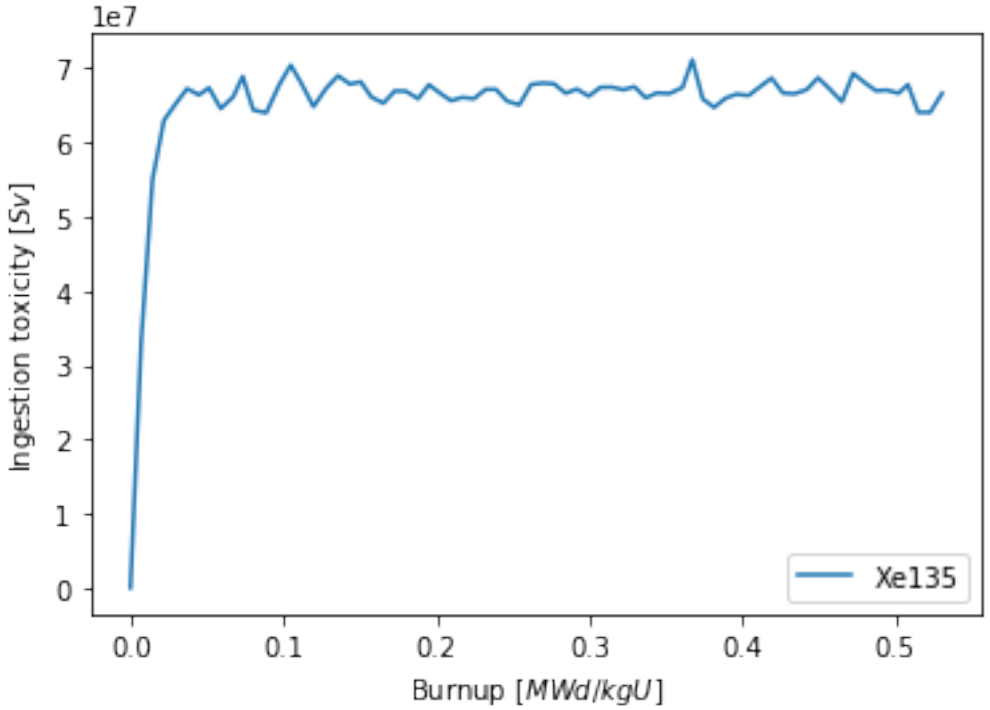
Specific days or values of burnup can be passed with the `timePoints` keyword. This will instruct the slicing tool to retrieve data that corresponds to values of `days` or `burnup` in the `timePoints` list. By default the method returns data for every time point on the material unless `timePoints` is given. Similarly, one can pass a string or list of strings as the `names` or `zai` arguments and obtain data for those specific isotopes. Data for every isotope is given if `names` or `zai` are not given.

```
>>> dayPoints = [0, 5, 10, 30]
>>> iso = ['Xe135', 'Sm149']
>>> zai = [541350, 621490]
>>> isoVals = fuel.getValues('days', 'a', dayPoints, iso)
>>> print(isoVals.shape)
>>> zaiVals = fuel.getValues('days', 'a', dayPoints, zai=zai)
print(isoVals - zaiVals)
(2, 4)
[[0.00000e+00 3.28067e+14 3.24606e+14 3.27144e+14]
 [0.00000e+00 0.00000e+00 0.00000e+00 0.00000e+00]]
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

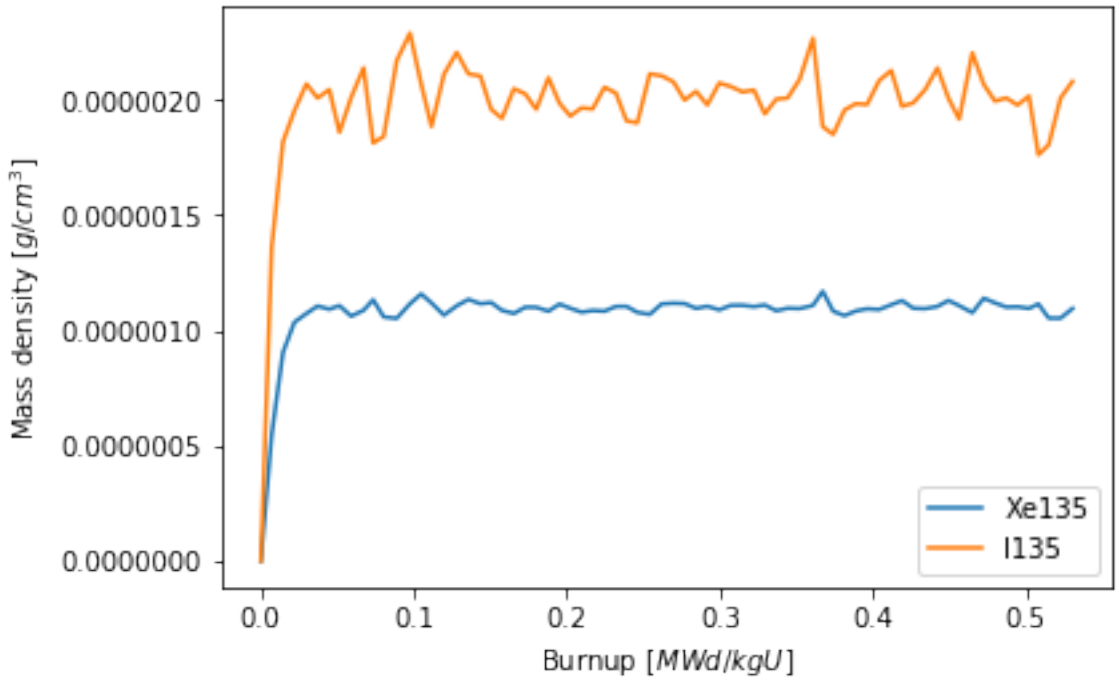
The *DepletedMaterial* uses this slicing for the built-in *plot()* method, which takes similar slicing arguments to `getValues()`.

By default, the plot method will plot data for all isotopes present, leading to very busy plots. The plots can be cleaned up by passing isotope names or `ZZAAAI` identifiers to the `names` or `zai` arguments, respectively.

```
>>> fuel.plot('burnup', 'ingTox', names='Xe135');
```



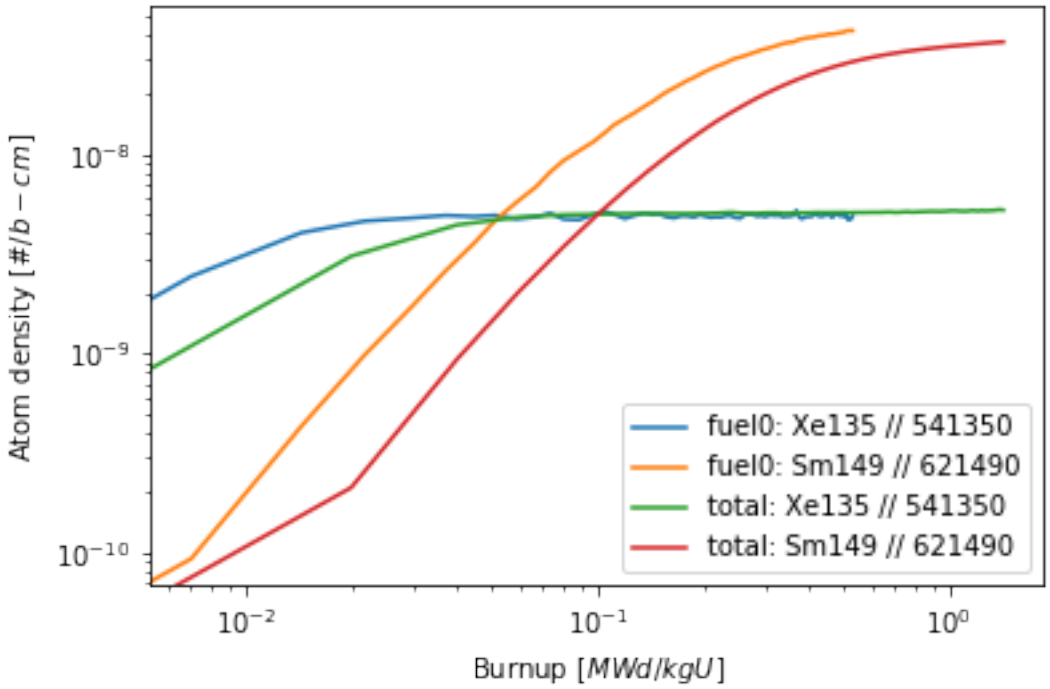

```
>>> fuel.plot('burnup', 'mdens', zai=[541350, 531350]);
```



This type of plotting can also be applied to the *DepletedMaterial* level, with similar options for formatting and retrieving data. The materials to be plotted can be filtered using the `materials` argument. The `labelFmt` argument can be used to apply a consistent label to each unique plot. This argument supports brace-delimited formatting, and will automatically replace strings like `{mat}` with the name of the material. The table below contains the special strings and their replacements

String	Replacement
`{mat}`	Name of the material
`{iso}`	Name of the isotope, e.g. `'U235'`
`{zai}`	ZZAAAI of the isotope, e.g. 922350

```
>>> dep.plot('burnup', 'adens', names=iso,
...          materials=['fuel0', 'total'],
...          labelFmt="{mat}: {iso} // {zai}", loglog=True);
```



4.4.4 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. The settings that control the depletion reader are

- *depletion.materials*
- *depletion.materialVariables*
- *depletion.metadataKeys*
- *depletion.processTotal*

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.

```
>>> rc['depletion.processTotal'] = False
>>> rc['depletion.metadataKeys'] = ['BU']
>>> rc['depletion.materialVariables'] = ['ADENS']
>>> rc['depletion.materials'] = [r'bglass\d+']
>>> bgReader = serpentTools.readDataFile(depFile)
```

(continues on next page)

(continued from previous page)

```
>>> bgReader.materials.keys()
dict_keys(['bglass0'])
>>> bglass = bgReader.materials['bglass0']
>>> bglass.data.keys()
dict_keys(['adens'])
```

4.4.5 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.

4.4.6 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

4.5 Sensitivity Reader

As of SERPENT 2.1.29, the ability to compute sensitivities using Generalized Perturbation Theory [gpt]. An overview of the functionality, and how to enable these features is included on the SERPENT wiki - Sensitivity Calculations. Sensitivity calculations produce `_sens.m` or `_sensN.m` files, depending on the version of SERPENT, and contain a collection of arrays and indexes, denoting the sensitivity of a quantity to perturbations in isotopic parameters, such as cross sections or fission spectrum. These perturbations can be applied to specific materials and/or isotopes.

The *SensitivityReader* is capable of reading this file and storing all the arrays and perturbation parameters contained therein. A basic plot method is also contained on the reader. .. note:

```
The preferred way to read your own output files is with the
|read-full| function. The |readData| function is used here
to make it easier to reproduce the examples
```

```
>>> import serpentTools
>>> sens = serpentTools.readDataFile('flattop_sens.m')
```

The arrays that are stored in *sensitivities* and *energyIntegratedSens* are stored under converted names. The original names from SERPENT are of the form `ADJ_PERT_KEFF_SENS` or `ADJ_PERT_KEFF_SENS_E_INT`, respectively. Since this reader stores the resulting arrays in unique locations, the names are converted to a succinct form. The two arrays listed above would be stored both as `keff` in *sensitivities* and *energyIntegratedSens*. All names are converted to `mixedCaseNames` to fit the style of the project.

These arrays are quite large, so only their shapes will be shown in this notebook.

```
>>> print(sens.sensitivities.keys(), sens.energyIntegratedSens.keys())
dict_keys(['keff']) dict_keys(['keff'])
>>> print(sens.sensitivities['keff'].shape)
(1, 2, 7, 175, 2)
>>> print(sens.energyIntegratedSens['keff'].shape)
(1, 2, 7, 2)
```

The energy grid structure and lethargy widths are stored on the reader, as *energies* and *lethargyWidths*.

```
>>> print(sens.energies.shape)
(176,)
>>> print(sens.energies[:10])
[1.00001e-11 1.00001e-07 4.13994e-07 5.31579e-07 6.82560e-07 8.76425e-07
1.12300e-06 1.44000e-06 1.85539e-06 2.38237e-06]
>>> print(sens.lethargyWidths.shape)
(175,)
>>> print(sens.lethargyWidths[:10])
[9.21034  1.42067  0.25     0.249999 0.250001 0.247908 0.248639 0.253452
0.250001 0.249999]
```

Ordered dictionaries *materials*, *zais*, and *perts* contain keys of the names of their respective data, and the corresponding index, `iSENS_ZAI_zzaaai`, in the sensitivity arrays. These arrays are zero-indexed, so the first item will have an index of zero. The data stored in the *sensitivities* and *energyIntegratedSens* dictionaries has the exact same structure as if the arrays were loaded into `MATLAB`/`Octave`, but with zero-indexing.

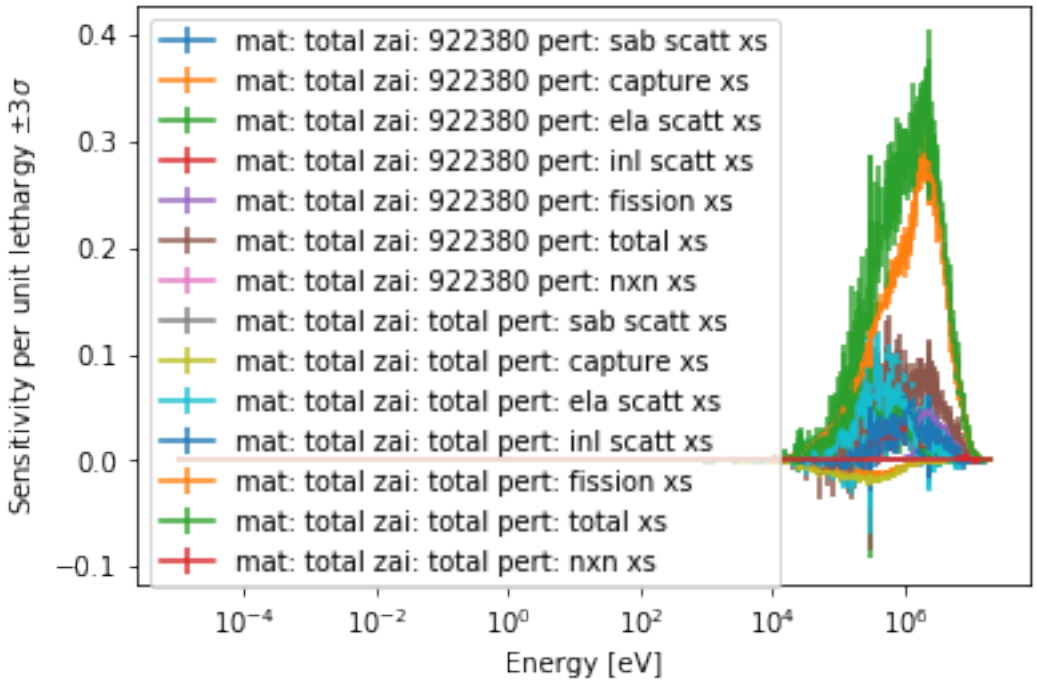
```
>>> print(sens.materials)
OrderedDict([('total', 0)])
>>> print(sens.zais)
OrderedDict([('total', 0), (922380, 1)])
>>> print(sens.perts)
OrderedDict([('total xs', 0), ('ela scatt xs', 1), ('sab scatt xs', 2), ('inl
scatt xs', 3), ('capture xs', 4), ('fission xs', 5), ('nxn xs', 6)])
```

4.5.1 Plotting

The *SensitivityReader* has a *plot()* method for visualizing the sensitivities.

Note: Without additional arguments, other than the name of the array, the *plot()* method will plot **all** permutations of materials, isotopes, and isotope perturbations present. This can lead to a very busy plot and legend, so it is recommended that additional arguments are passed.

```
>>> sens.plot('keff');
```



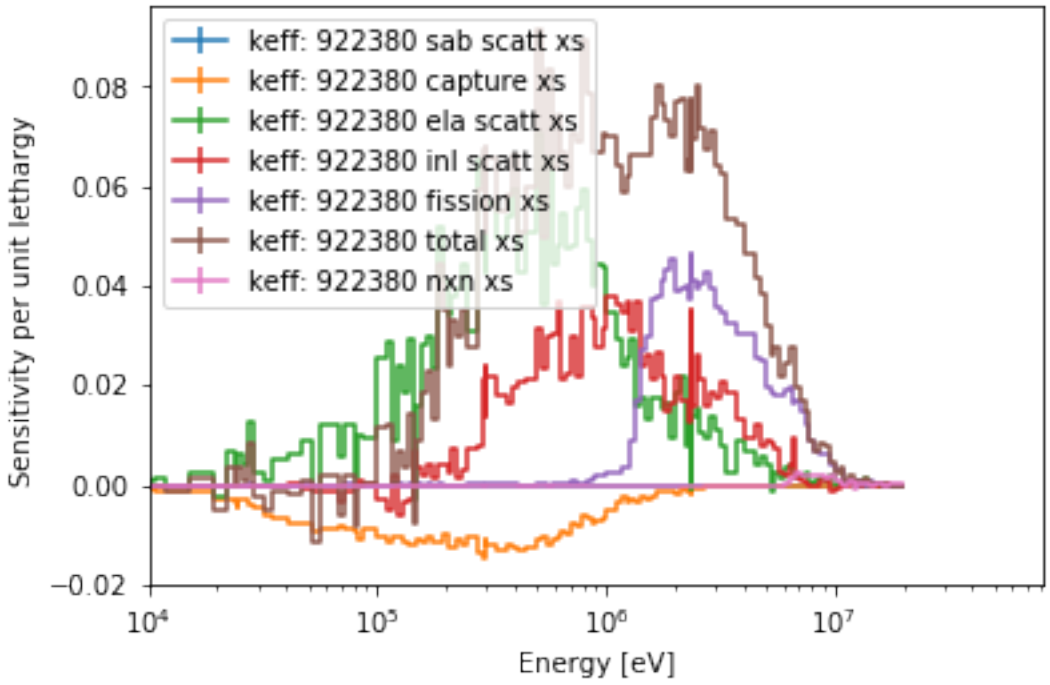
The following arguments can be used to filter the data present:

key	Action
`'zai'`	Isotopes(s) of interest
`'pert'`	Perturbation(s) of interest
`'mat'`	Material(s) of interest

The `sigma` argument can be used to adjust the confidence interval applied to the plot. The `labelFmt` argument can be used to modify the label used for each plot. The following replacements will be made:

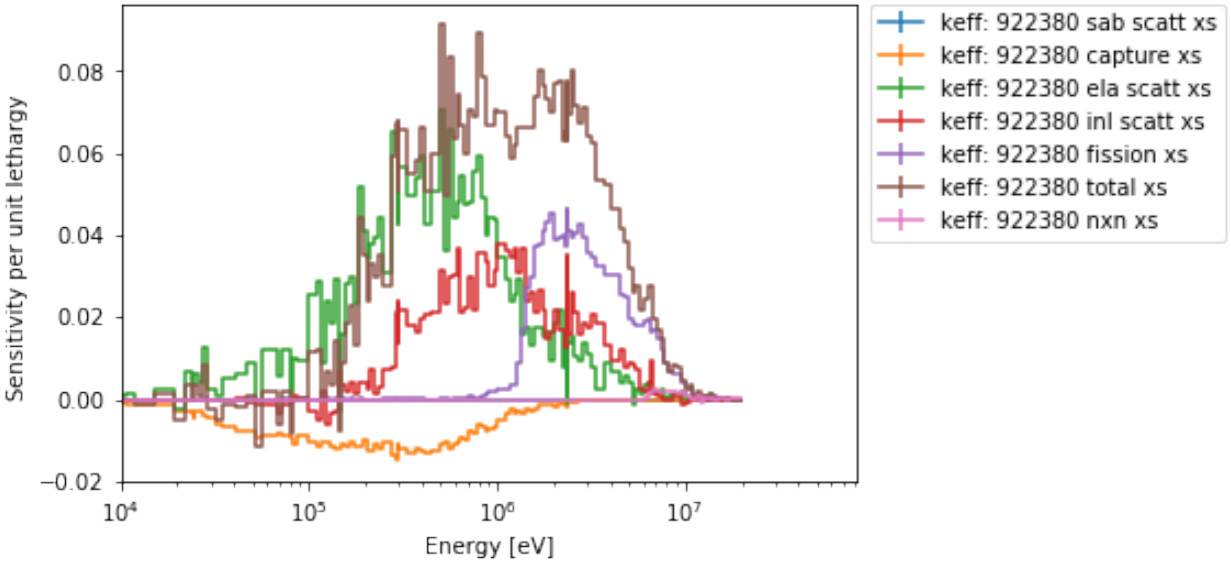
1. `{r}` - name of the response being plotted 1. `{m}` - name of the material 1. `{z}` - isotope zai 1. `{p}` - specific perturbation

```
>>> ax = sens.plot('keff', 922380, mat='total', sigma=0,
...                labelFmt="{r}: {z} {p}")
>>> ax.set_xlim(1E4);  # set the lower limit to be closer to what we care about
```

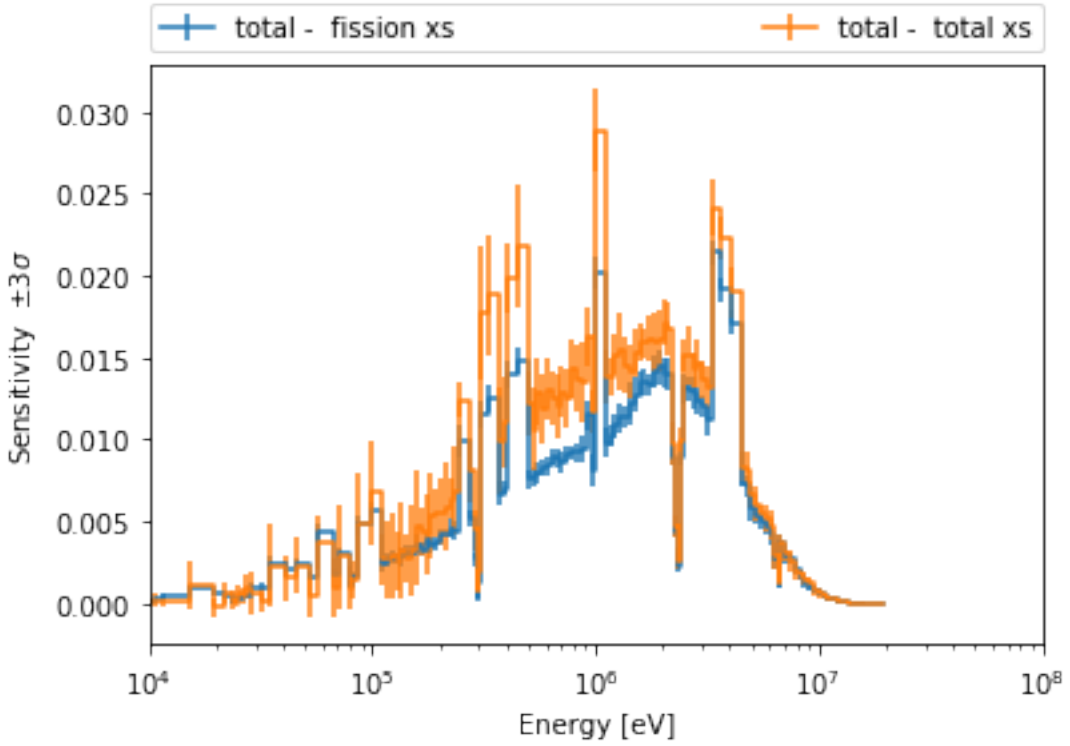


The argument `normalize` is used to turn on/off normalization per unit lethargy, while `legend` can be used to turn off the legend, or set the legend outside the plot.

```
>>> ax = sens.plot('keff', 922380, mat='total', sigma=0,
...                labelFmt="{r}: {z} {p}", legend='right')
>>> ax.set_xlim(1E4);  # set the lower limit to be closer to what we care about
```




```
>>> sens.plot('keff', zai='total', pert=['total xs', 'fission xs'], labelFmt="{z} -
↪{p}",
...           legend='above', ncol=2, normalize=False)
>>> pyplot.xlim(1E4, 1E8);
```



4.5.2 Conclusion

The *SensitivityReader* can quickly read sensitivity files, and stores all data present in the file. A versatile *plot()* method can be used to quickly visualize sensitivities.

4.6 Cross Section Reader/Plotter

4.6.1 Basic Operation

Firstly, to get started plotting some cross sections from Serpent, you generate a `yourInputFileName_xs.m` file using set xsplot as documented on the Serpent wiki. `serpentTools` can then read the output, figuring out its file type automatically as with other readers. Let's plot some data used in the `serpentTools` regression suite.

Note: The preferred way to read your own output files is with the *serpentTools.read()* function. The `serpentTools.readDataFile()` function is used here to make it easier to reproduce the examples

---

```
>>> import serpentTools
>>> xsreader = serpentTools.readDataFile('plut_xs0.m')
```

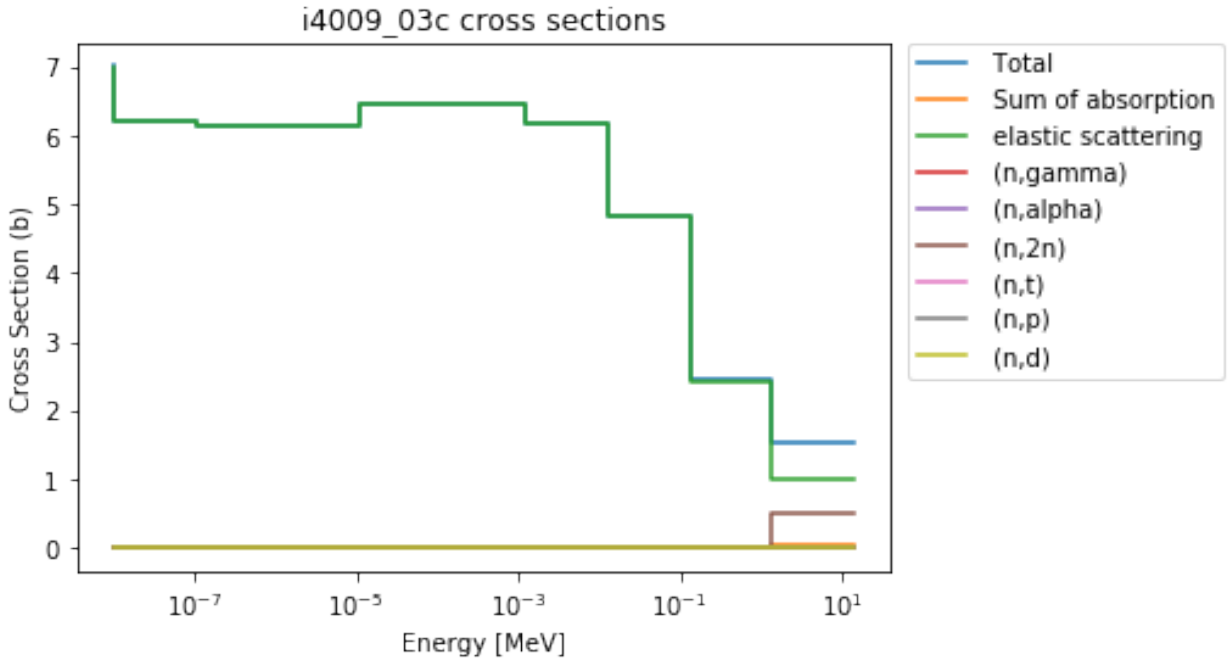
This file contains some cross sections from a Serpent case containing a chunk of plutonium metal reflected by beryllium. Let's see what cross sections are available from the file:

```
>>> xsreader.xsections.keys()
dict_keys(['i4009_03c', 'i7014_03c', 'i8016_03c', 'i94239_03c', 'mbe',
'mfissile'])
```

Notice that the important part of the reader is the *xsections* attribute, which contains a dictionary of named XSData objects. Entries starting with "i" are isotopes, while "m" preceded names are materials. Notably, materials not appearing in the neutronics calculation, e.g., external tanks in Serpent continuous reprocessing calculations, are not printed in the yourInputFileName_xs.m file.

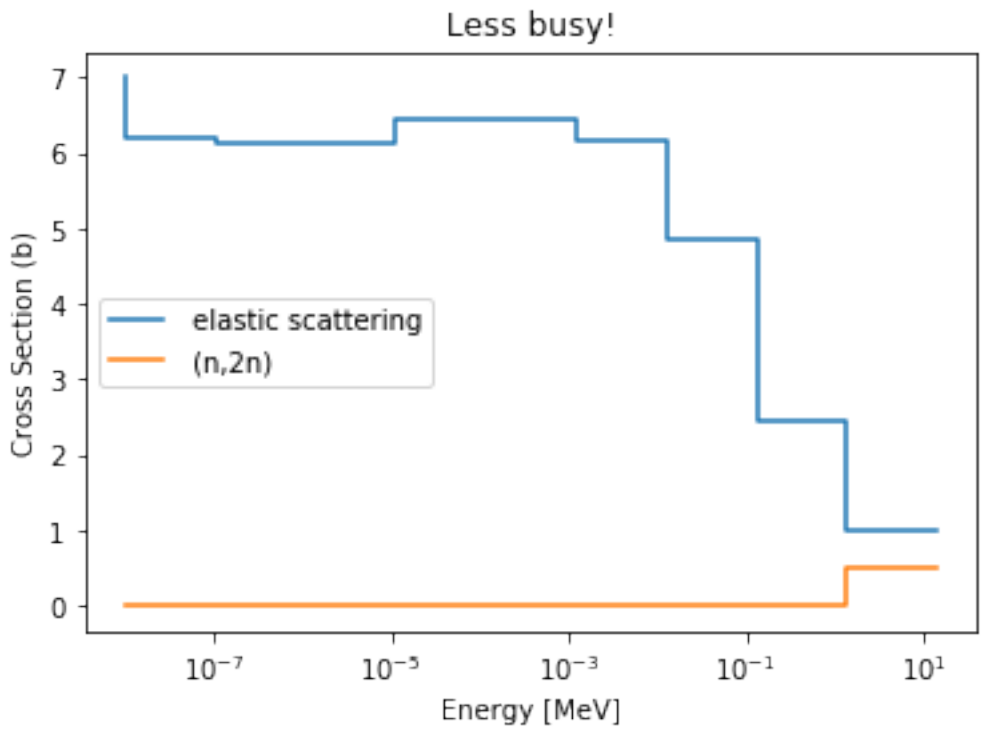
Plotting the entries is very easy, check this out:

```
>>> xsreader.xsections['i4009_03c'].plot(legend='right');
```



This is nice to have an automatically generated legend, but gets somewhat busy quickly. So, it's easy to check which MT numbers are available, and plot only a few:

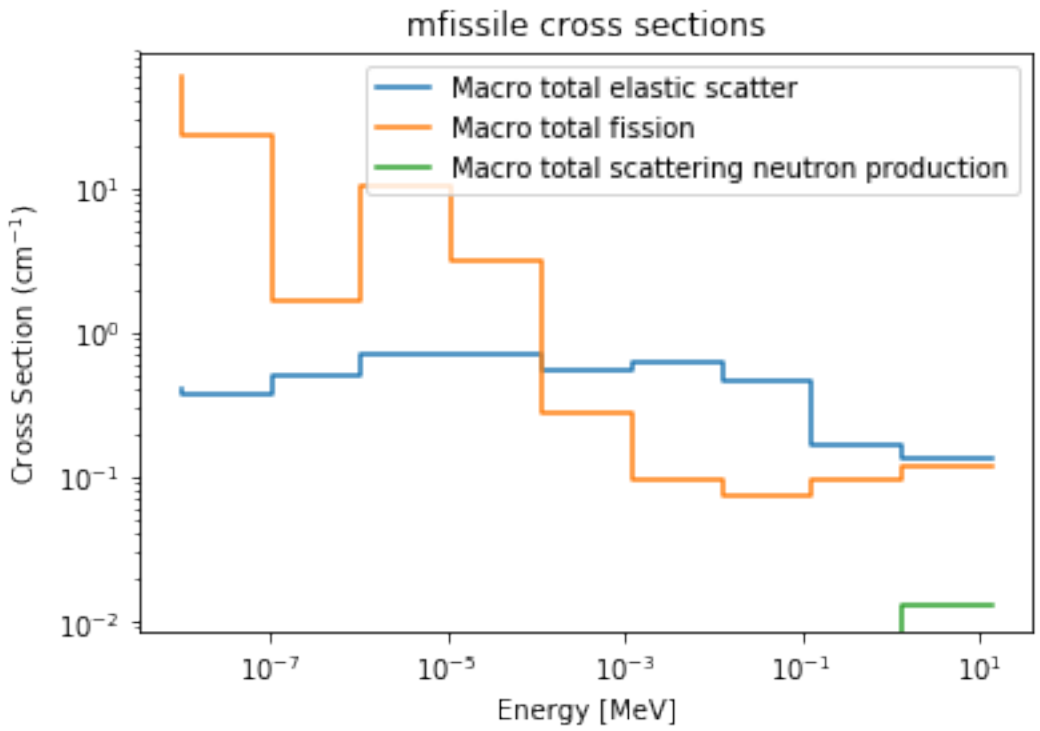
```
>>> xsreader.xsections['i4009_03c'].showMT()
MT numbers available for i4009_03c:
-----------------------------------
1     Total
101   Sum of absorption
2     elastic scattering
102   (n,gamma)
107   (n,alpha)
16    (n,2n)
105   (n,t)
103   (n,p)
104   (n,d)
>>> xsreader.xsections['i4009_03c'].plot(mts=[2, 16], title='Less busy!');
```



Of course, the same process can be applied to materials, but Serpent has some special unique negative MT numbers. The code will give you their meaning without requiring your reference back to the wiki.

```
>>> xsreader.xsections['mfissile'].showMT()
MT numbers available for mfissile:
-----------------------------------
-1   Macro total
-3   Macro total elastic scatter
-2   Macro total capture
-6   Macro total fission
-7   Macro total fission neutron production
-16  Macro total scattering neutron production

>>> xsreader.xsections['mfissile'].plot(mts=[-3, -6, -16], loglog=True)
```



Lastly, there are two ways to directly access data from Serpent's xsplot output. Firstly, if you'd like to see the data as-stored, just check the attribute called "data" belonging to the `XSData` object. Its columns correspond to MT reactions, ordered in the same way as *`showMT()`* presents. The rows correspond to values at the energies in metadata['egrid'].

The other method regards presenting tabular data in a visually appealing way. It uses pandas though, so make sure you have that installed to obtain pretty tables.

```
>>> xsreader.xsections['mfissile'].xsdata
array([[7.84253e+01, 4.04950e-01, 1.96698e+01, 5.83505e+01, 1.67674e+02,
        0.00000e+00],
       [3.61666e+01, 3.69643e-01, 1.20450e+01, 2.37520e+01, 6.80558e+01,
        0.00000e+00],
       [2.54417e+00, 5.06089e-01, 4.10559e-01, 1.62752e+00, 4.67294e+00,
        0.00000e+00],
       [1.30654e+01, 7.15384e-01, 2.01598e+00, 1.03340e+01, 2.95250e+01,
        0.00000e+00],
       [4.27811e+00, 7.21668e-01, 4.34122e-01, 3.12232e+00, 9.00007e+00,
        0.00000e+00],
       [8.22536e-01, 5.37059e-01, 3.51405e-03, 2.81963e-01, 8.14254e-01,
        0.00000e+00],
       [7.81066e-01, 6.23379e-01, 4.77288e-02, 9.38536e-02, 2.71066e-01,
        0.00000e+00],
       [5.83509e-01, 4.58020e-01, 1.08053e-02, 7.51650e-02, 2.17468e-01,
        0.00000e+00],
       [3.41750e-01, 1.63555e-01, 7.72110e-04, 9.51295e-02, 2.91685e-01,
        0.00000e+00],
       [2.93887e-01, 1.36424e-01, 1.13642e-04, 1.20609e-01, 5.96505e-01,
        1.28477e-02]])
>>> xsreader.xsections['mfissile'].tabulate()
```

4.6.2 Conclusions

`serpentTools` can plot your Serpent XS data in a friendly way. We're always looking to improve the feel of the code though, so let us know if there are changes you would like.

Keep in mind that setting an energy grid with closer to 10000 points makes far prettier XS plots however. There were none in this example to not clog up the repository.

4.7 Results Reader

4.7.1 Basic operations

This notebook demonstrates the capabilities of the serpentTools to read Serpent results files. SERPENT [serpent] produces a result file (i.e. `_res.m`), containing general results (e.g. `k-eff`), metadata (e.g. `title`) and homogenized cross-sections. The homogenized cross-section sets are printed in the results file for all the requested universes. The *`ResultsReader`* is capable of reading this file, and storing the data inside *`HomogUniv`* objects. Each such object has methods and attributes that should ease the analyses.

Note: The preferred way to read your own output files is with the *`serpentTools.read()`* function. The `serpentTools.readDataFile()` function is used here to make it easier to reproduce the examples

---

```
>>> import numpy as np
>>> import serpentTools
>>> from serpentTools.settings import rc

>>> resFile = 'InnerAssembly_res.m'
>>> res = serpentTools.readDataFile(resFile)
```

Metadata

`metadata` is a collective data that describes the problem. These are values that do not change over burnup and across homogenized universes. The following data is included: titles, data paths, and other descriptive data exist on the reader

```
>>> print(res.metadata['version'])
Serpent 2.1.30
>>> print(res.metadata['decayDataFilePath'])
/nv/hp22/dkotlyar6/data/Codes/DATA/endfb7/sss_endfb7.dec
>>> print(res.metadata['inputFileName'])
InnerAssembly
>>> res.metadata.keys()
dict_keys(['version', 'compileDate', 'debug', 'title', 'confidentialData',
'inputFileName', 'workingDirectory', 'hostname', 'cpuType', 'cpuMhz',
'startDate', 'completeDate', 'pop', 'cycles', 'skip', 'batchInterval',
'srcNormMode', 'seed', 'ufsMode', 'ufsOrder', 'neutronTransportMode',
'photonTransportMode', 'groupConstantGeneration', 'b1Calculation',
'b1BurnupCorrection', 'implicitReactionRates', 'optimizationMode',
'reconstructMicroxs', 'reconstructMacroxs', 'doubleIndexing', 'mgMajorantMode',
'spectrumCollapse', 'mpiTasks', 'ompThreads', 'mpiReproducibility',
'ompReproducibility', 'ompHistoryProfile', 'shareBufArray', 'shareRes2Array',
'xsDataFilePath', 'decayDataFilePath', 'sfyDataFilePath', 'nfyDataFilePath',
'braDataFilePath'])
```

(continues on next page)

(continued from previous page)

```
>>> res.metadata['startDate']
'Sat Apr 28 06:09:54 2018'
>>> res.metadata['pop'], res.metadata['skip']  , res.metadata['cycles']
(5000, 10, 50)
```

Results Data

Results are stored as a function of time/burnup/index and include integral parameters of the system. Results, such as `k-eff`, total `flux`, and execution times are included in *`resdata`*. Some results include values and uncertainties (e.g. criticality) and some just the values (e.g. CPU resources).

```
>>> list(res.resdata.keys())[0:5]
['minMacroxs', 'dtThresh', 'stFrac', 'dtFrac', 'dtEff']
```

Values are presented in similar fashion as if they were read in to Matlab, with one exception. Serpent currently appends a new row for each burnup step, but also for each set of homogenized universe. This results in repetition of many quantities as Serpent loops over group constant data. The *`ResultsReader`* understands Serpent outputs and knows when to append "new" data to avoid repetition.

The structure of the data is otherwise identical to Matlab. For many quantities, the first column contains expected values, and the second column contains relative uncertainties.

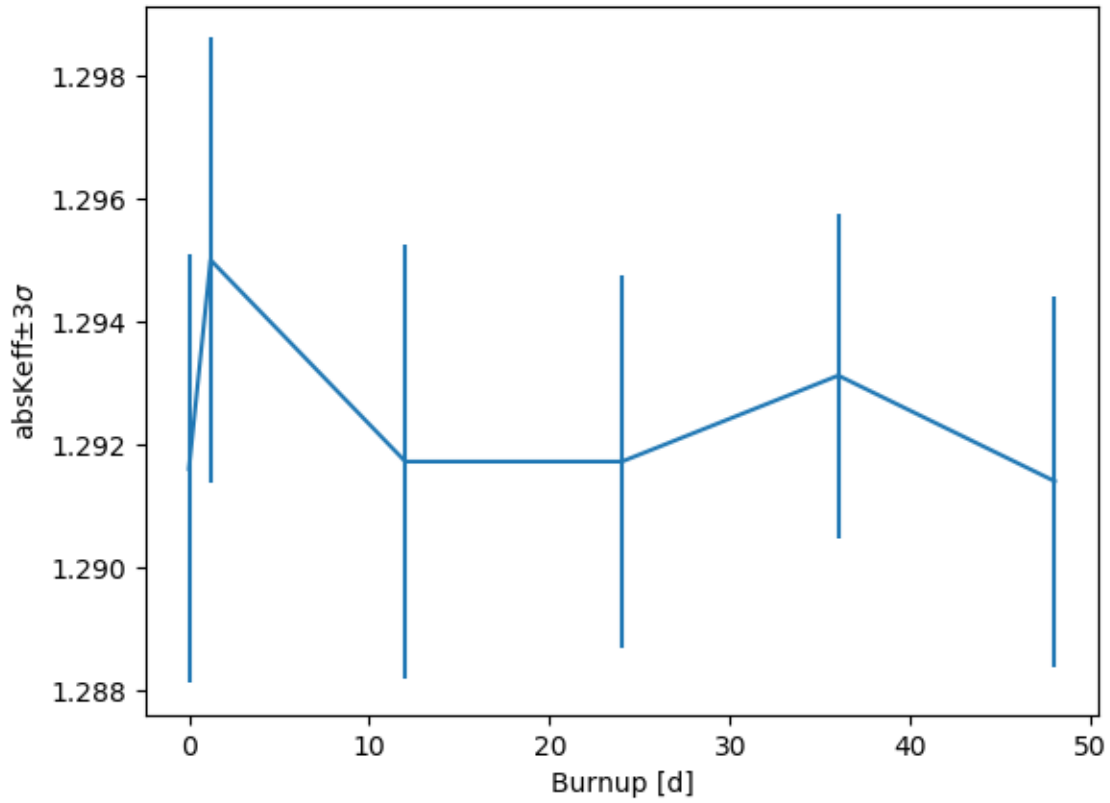
```
>>> res.resdata['absKeff'])
array([[  1.29160000e+00, 9.00000000e-04],
       [  1.29500000e+00, 9.30000000e-04],
       [  1.29172000e+00, 9.10000000e-04],
       [  1.29172000e+00, 7.80000000e-04],
       [  1.29312000e+00, 6.80000000e-04],
       [  1.29140000e+00, 7.80000000e-04]]
>>> res.resdata['absKeff'][:,0]
array([ 1.2916 ,  1.295  ,  1.29172,  1.29172,  1.29312,  1.2914 ])

>>> res.resdata['burnup']
array([[ 0.      , 0.      ],
       [ 0.1     , 0.100001],
       [ 1.      , 1.00001 ],
       [ 2.      , 2.00001 ],
       [ 3.      , 3.00003 ],
       [ 4.      , 4.00004 ]]
>>> res.resdata['burnDays']
[[  0.     ],
 [  1.20048],
 [ 12.0048 ],
 [ 24.0096 ],
 [ 36.0144 ],
 [ 48.0192 ]]
```

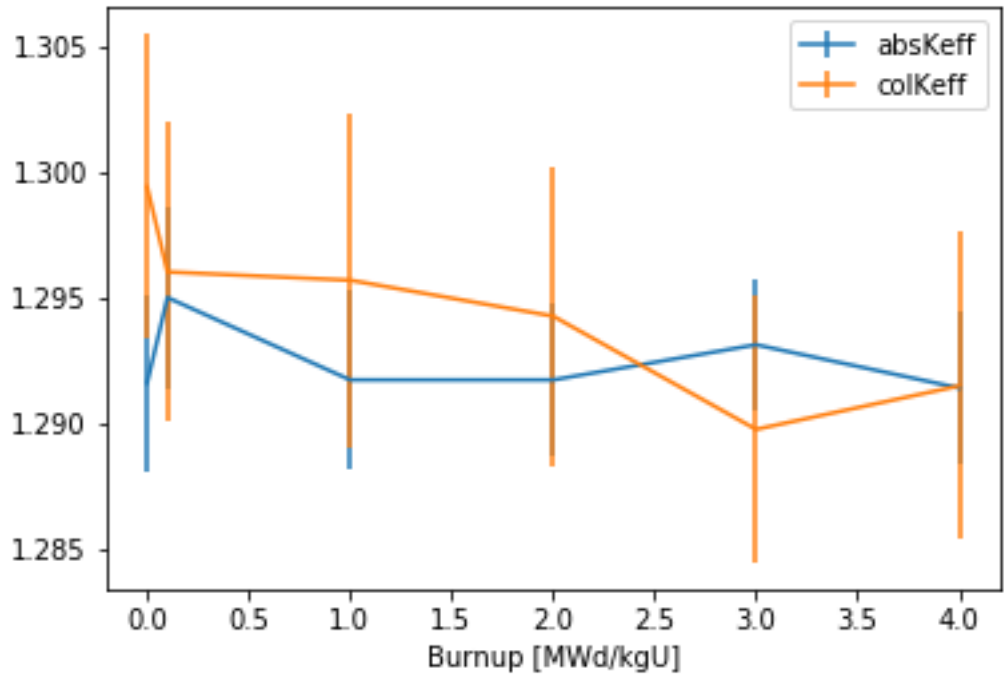
4.7.2 Plotting Results Data

The *ResultsReader* has a versatile *plot()* method, used to plot primary time-dependent data from the result file. One can plot data from one or more quantities against various metrics of time. Control over axis formatting, legend placement, and label formatting is easily yielded to the user.

```
>>> res.plot('absKeff')
```

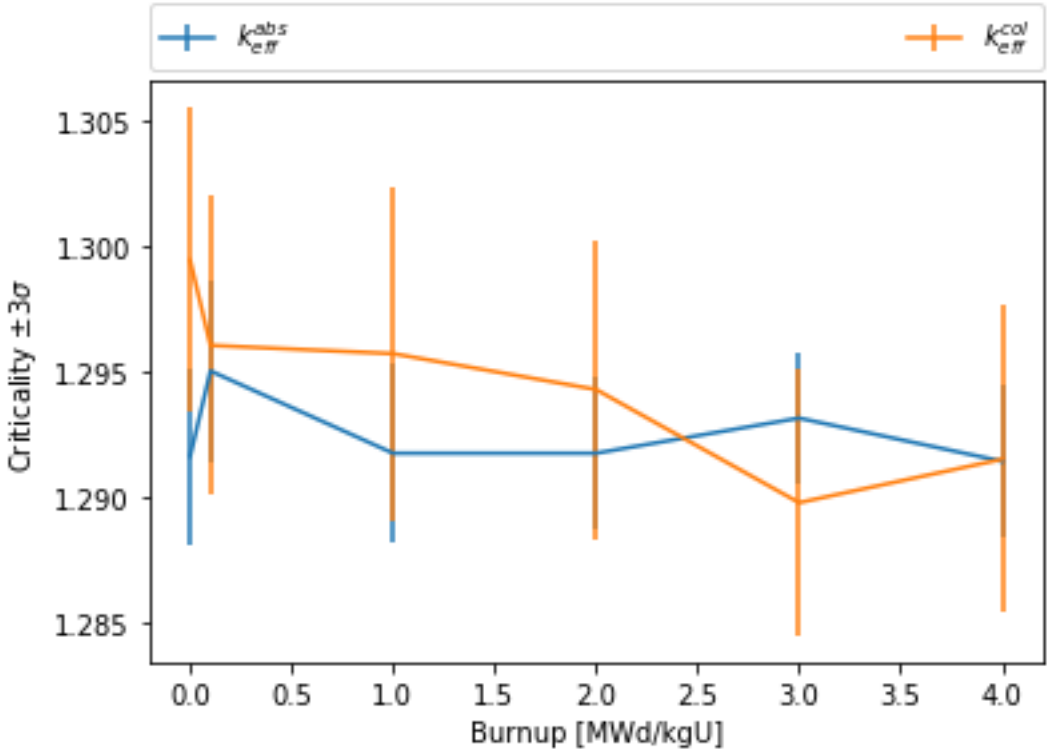



```
>>> res.plot('burnup', ['absKeff', 'colKeff'])
```



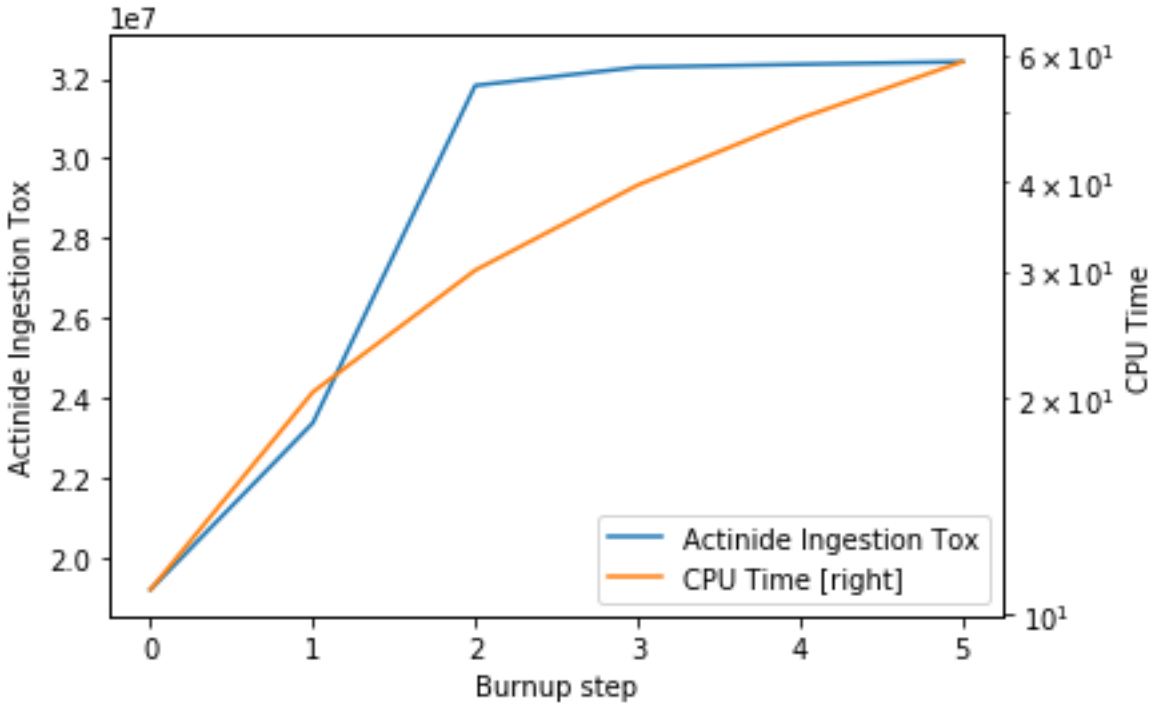
Pass a dictionary of `variable: label` pairs to set plot labels.

```
>>> res.plot(
>>>   'burnup', {
>>>       'absKeff': '$k_{eff}^{abs}$',
>>>       'colKeff': '$k_{eff}^{col}$',},
>>>   ylabel=r'Criticality $\pm 3\sigma$',
>>>   legend='above', ncol=2,
>>>   )
```



Using the `right` argument, quantities can be plotted using the left and right y-axis. Similar formatting options are available.

```
>>> res.plot(
>>>   'burnStep',
>>>   {'actinideIngTox': 'Actinide Ing. Tox'},
>>>   right={'totCpuTime': 'CPU Time [right]'},
>>>   sigma=0, rightlabel="CPU Time",
>>>   # set the yscale to log only of right axis
>>>   logy=[False, True],
>>>   )
```



Universe Data

Universe data is stored for each state point in the *universes* dictionary. Keys are `UnivTuple` representing `('univ',burnup, burnupIdx, time)`

- `'univ'`: universe ID (e.g., `'0'`)
- `burnup`: in MWd/kg,
- `burnupIdx`: step index
- `time`: in days

and can be indexed by attribute or by position.

```
>>> for key in sorted(res.universes):
...     break
>>> key
UnivTuple(universe='0', burnup=0.0, step=0, days=0.0)
>>> key[0]
'0'
>>> key.burnup == key[1]
True
```

Results, such as infinite cross-sections, b1-leakage corrected cross-sections, kinetic parameters, are included in *universes*. All the results include values and uncertainties.

```
>>> res.universes.keys()
dict_keys([
    UnivTuple(universe='3101', burnup=0.0, step=0, days=0.0),
    UnivTuple(universe='3102', burnup=0.0, step=0, days=0.0),
    UnivTuple(universe='0', burnup=0.0, step=0, days=0.0),
```

(continues on next page)

(continued from previous page)

```
    UnivTuple(universe='3101', burnup=0.1, step=1, days=1.20048),
    UnivTuple(universe='3102', burnup=0.1, step=1, days=1.20048),
    ...
    UnivTuple(universe='3101', burnup=4.0, step=5, days=48.0192),
    UnivTuple(universe='3102', burnup=4.0, step=5, days=48.0192),
    UnivTuple(universe='0', burnup=4.0, step=5, days=48.0192)])
```

One can directly index into *universes* to obtain a specific *HomogUniv* object:

```
>>> print(res.universes['0', 0, 0, 0])
<HomogUniv 0: burnup: 0.00000E+00 MWd/kgU, 0.00000E+00 days>
```

However, this requires knowledge of all four parameters, which may be difficult. The *getUniv()* method retrieves the first universe that matches a set of parameters, the universe and at least one point in time. While all four identifiers (universe id, burnup, step, and time) can be provided, the latter three are usually redundant.

```
>>> univ0 = res.getUniv('0', timeDays=24.0096)
>>> print(univ0)
<HomogUniv 0: burnup: 0.00000E+00 MWd/kgU, 0.00000E+00 days>
>>> univ3101 = res.getUniv('3101', index=3)
>>> print(univ3101)
<HomogUniv 3101: burnup: 2.00000E+00 MWd/kgU, 2.40096E+01 days>
>>> univ3102 = res.getUniv('3102', burnup=0.1)
>>> print(univ3102)
<HomogUniv 3102: burnup: 1.00000E-01 MWd/kgU, 1.20048E+00 days>
```

4.7.3 Working with homogenized universe data

Each state contains the same data fields, which can be obtained by using the following attributes on the *HomogUniv* object:

- *infExp*: infinite values, e.g. `INF_ABS`
- *infUnc*: infinite uncertainties
- *b1Exp*: b1 (leakage corrected) values, e.g. `B1_ABS`
- *b1Unc*: b1 (leakage corrected) uncertainties
- *gc*: variables that are not included in `inf` or `b1`, e.g. `BETA`
- *gcUnc*: uncertainties for quantities in *gc*
- *groups*: macro energy group structure, MeV
- *microGroups*: micro energy group structure, MeV

The parser reads all the variables by default. The values are all energy dependent in the order they would appear in the results file:

```
>>> univ0.infExp.keys()
dict_keys(['infMicroFlx', 'infKinf', 'infFlx', 'infFissFlx', 'infTot',
'infCapt', 'infAbs', 'infFiss', 'infNsf', 'infNubar', 'infKappa', 'infInvv',
'infScatt0', 'infScatt1', 'infScatt2', 'infScatt3', 'infScatt4', 'infScatt5',
'infScatt6', 'infScatt7', 'infScattp0', 'infScattp1', 'infScattp2',
'infScattp3', 'infScattp4', 'infScattp5', 'infScattp6', 'infScattp7',
'infTranspxs', 'infDiffcoef', 'infRabsxs', 'infRemxs', 'infI135Yield',
'infXe135Yield', 'infPm147Yield', 'infPm148Yield', 'infPm148mYield',
```

(continues on next page)

(continued from previous page)

```
'infPm149Yield', 'infSm149Yield', 'infI135MicroAbs', 'infXe135MicroAbs',
'infPm147MicroAbs', 'infPm148MicroAbs', 'infPm148mMicroAbs',
'infPm149MicroAbs', 'infSm149MicroAbs', 'infXe135MacroAbs', 'infSm149MacroAbs',
'infChit', 'infChip', 'infChid', 'infS0', 'infS1', 'infS2', 'infS3', 'infS4',
'infS5', 'infS6', 'infS7', 'infSp0', 'infSp1', 'infSp2', 'infSp3', 'infSp4',
'infSp5', 'infSp6', 'infSp7'])

>>> univ0.infExp['infAbs']
array([ 0.0170306 ,  0.0124957 ,  0.00777066,  0.00773255,  0.00699608,
        0.00410746,  0.00334604,  0.00296948,  0.0030725 ,  0.00335412,
        0.00403133,  0.00506587,  0.00651475,  0.00737292,  0.00907442,
        0.0113446 ,  0.0125896 ,  0.0164987 ,  0.0181642 ,  0.0266464 ,
        0.0292439 ,  0.0315338 ,  0.0463069 ,  0.0807952 ])

>>> univ0.infExp['infFlx']
array([  1.10460000e+15,   1.72386000e+16,   7.78465000e+16,
         1.70307000e+17,   2.85783000e+17,   4.61226000e+17,
         8.04999000e+17,   1.17536000e+18,   1.17488000e+18,
         1.26626000e+18,   1.03476000e+18,   7.58885000e+17,
         4.95687000e+17,   5.85369000e+17,   2.81921000e+17,
         1.16665000e+17,   8.06833000e+16,   2.26450000e+16,
         6.51541000e+16,   2.79929000e+16,   8.87468000e+15,
         1.70822000e+15,   8.87055000e+14,   6.22266000e+13])
```

Uncertainties can be obtained in a similar was by using the *infUnc* field:

```
>>> univ0.infUnc['infFlx']
array([0.02125, 0.0287 , 0.00901, 0.00721, 0.00441, 0.00434, 0.00448,
       0.0007 , 0.00369, 0.00071, 0.00045, 0.00133, 0.00061, 0.00341,
       0.00674, 0.00197, 0.00802, 0.00368, 0.00127, 0.00046, 0.02806,
       0.0491 , 0.19529, 0.16476])
```

Serpent also outputs the `B1` cross-sections. However, the user must enable the `B1` option by setting the `fum` card: http://serpent.vtt.fi/mediawiki/index.php/Input_syntax_manual#set_fum If this card is not enabled by the user, the `B1_` variables will all be zeros.

```
>>> univ0.b1Exp.keys()
dict_keys(['b1MicroFlx', 'b1Kinf', 'b1Keff', 'b1B2', 'b1Err', 'b1Flx',
'b1FissFlx', 'b1Tot', 'b1Capt', 'b1Abs', 'b1Fiss', 'b1Nsf', 'b1Nubar',
'b1Kappa', 'b1Invv', 'b1Scatt0', 'b1Scatt1', 'b1Scatt2', 'b1Scatt3',
'b1Scatt4', 'b1Scatt5', 'b1Scatt6', 'b1Scatt7', 'b1Scattp0', 'b1Scattp1',
'b1Scattp2', 'b1Scattp3', 'b1Scattp4', 'b1Scattp5', 'b1Scattp6', 'b1Scattp7',
'b1Transpxs', 'b1Diffcoef', 'b1Rabsxs', 'b1Remxs', 'b1I135Yield',
'b1Xe135Yield', 'b1Pm147Yield', 'b1Pm148Yield', 'b1Pm148mYield',
'b1Pm149Yield', 'b1Sm149Yield', 'b1I135MicroAbs', 'b1Xe135MicroAbs',
'b1Pm147MicroAbs', 'b1Pm148MicroAbs', 'b1Pm148mMicroAbs', 'b1Pm149MicroAbs',
'b1Sm149MicroAbs', 'b1Xe135MacroAbs', 'b1Sm149MacroAbs', 'b1Chit', 'b1Chip',
'b1Chid', 'b1S0', 'b1S1', 'b1S2', 'b1S3', 'b1S4', 'b1S5', 'b1S6', 'b1S7',
'b1Sp0', 'b1Sp1', 'b1Sp2', 'b1Sp3', 'b1Sp4', 'b1Sp5', 'b1Sp6', 'b1Sp7'])

>>> univ3101.b1Exp['b1Flx']
array([  1.20660000e+15,   1.65202000e+16,   7.47956000e+16,
         1.62709000e+17,   2.74814000e+17,   4.22295000e+17,
         7.04931000e+17,   9.70795000e+17,   9.11899000e+17,
         9.33758000e+17,   7.23255000e+17,   5.00291000e+17,
         3.16644000e+17,   3.52049000e+17,   1.62308000e+17,
```

(continues on next page)

(continued from previous page)

```
         6.68674000e+16,   4.47932000e+16,   1.23599000e+16,
         3.51299000e+16,   1.46504000e+16,   4.38516000e+15,
         7.96971000e+14,   3.54233000e+14,   2.11013000e+13])
```

Data that does not contain the prefix `INF_` or `B1_` is stored under the *gc* and *gcUnc* fields. Criticality, kinetic, and other variables are stored under this field.

```
>>> univ3101.gc.keys()
dict_keys(['cmmTranspxs', 'cmmTranspxsX', 'cmmTranspxsY', 'cmmTranspxsZ',
'cmmDiffcoef', 'cmmDiffcoefX', 'cmmDiffcoefY', 'cmmDiffcoefZ', 'betaEff',
'lambda'])
>>> univ3101.gc['betaEff']
array([  3.04272000e-03,   8.93131000e-05,   6.59324000e-04,
         5.62858000e-04,   1.04108000e-03,   5.67326000e-04,
         1.22822000e-04])
```

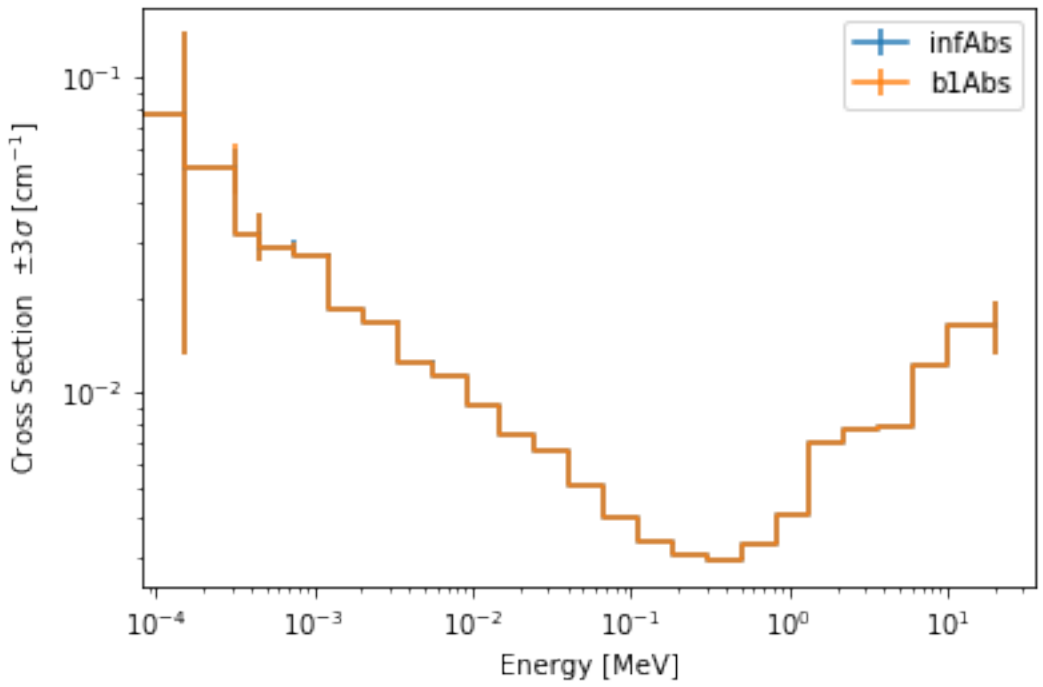
Macro- and micro- group structures are stored directly on the universe in MeV as they appear in Serpent output files. This means that the macro-group structure is in order of descending energy, while micro-group are in order of increasing energy:

```
>>> univ3101.groups
array([  1.00000000e+37,   1.00000000e+01,   6.06530000e+00,
         3.67880000e+00,   2.23130000e+00,   1.35340000e+00,
         8.20850000e-01,   4.97870000e-01,   3.01970000e-01,
         1.83160000e-01,   1.11090000e-01,   6.73800000e-02,
         4.08680000e-02,   2.47880000e-02,   1.50340000e-02,
         9.11880000e-03,   5.53090000e-03,   3.35460000e-03,
         2.03470000e-03,   1.23410000e-03,   7.48520000e-04,
         4.54000000e-04,   3.12030000e-04,   1.48940000e-04,
         0.00000000e+00])
>>> univ3101.microGroups[:5:]
array([  1.00000000e-10,   1.48940000e-04,   1.65250000e-04,
         1.81560000e-04,   1.97870000e-04])
```

4.7.4 Plotting universes

HomogUniv objects can plot group constants using their *plot()* method. This method has a range of formatting options, with defaults corresponding to plotting macroscopic cross sections. This is manifested in the default y axis label, but can be easily adjusted.

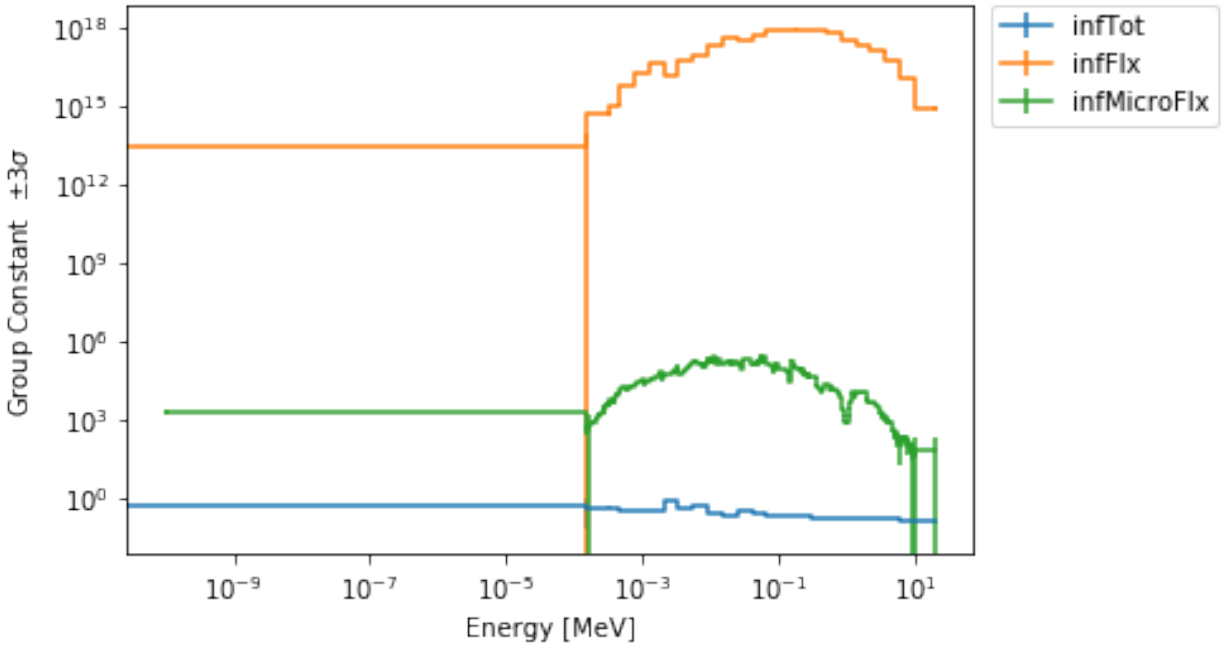
```
>>> univ3101.plot(['infAbs', 'b1Abs']);
```



Macroscopic and microscopic quantities, such as micro-group flux, can be plotted on the same figure.

Note: The units and presentation of the micro- and macro-group fluxes are dissimilar, and the units do not agree with that of the assumed group constants. This will adjust the default y-label, as demonstrated below.

```
>>> univ3101.plot(['infTot', 'infFlx', 'infMicroFlx'], legend='right');
```

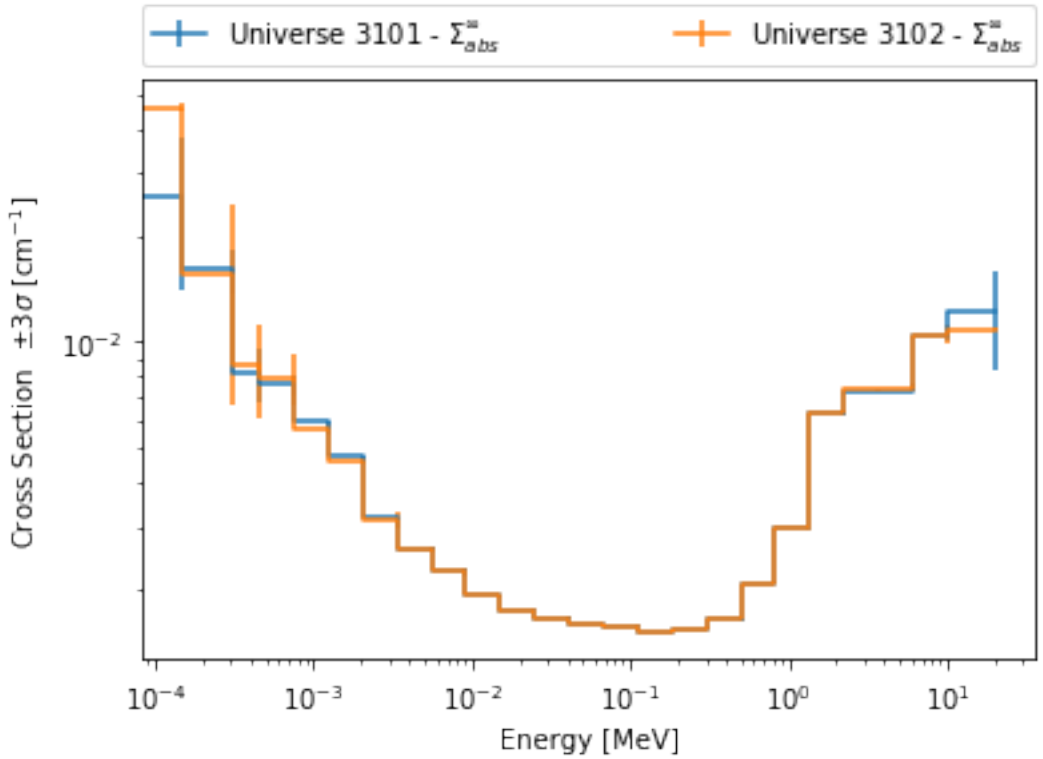


For plotting data from multiple universes, pass the returned `matplotlib.axes.Axes` object, on which the plot was drawn, into the plot method for the next universe. The `labelFmt` argument can be used to differentiate between plotted data. The following strings are replaced when creating the labels:

String	Replaced value
`{k}`	Name of variable plotted
`{u}`	Name of this universe
`{b}`	Value of burnup in MWd/kgU
`{d}`	Value of burnup in days
`{i}`	Burnup index

These can be used in conjunction with the LaTeX rendering system .

```
>>> fmt = r"Universe {u} - $\Sigma_{abs}^\infty$"
>>> ax = univ3101.plot('infFiss', labelFmt=fmt)
>>> univ3102.plot('infFiss', ax=ax, labelFmt=fmt, legend='above', ncol=2);
```



4.7.5 User Defined Settings

The user is able to filter the required information by using the *`serpentTools.settings.rc`* settings object. A detailed description on how to use the settings can be found on: *Default Settings*.

```
>>> from serpentTools.settings import rc
>>> rc.keys()
dict_keys(['branching.areUncsPresent', 'branching.intVariables',
'branching.floatVariables', 'depletion.metadataKeys',
'depletion.materialVariables', 'depletion.materials', 'depletion.processTotal',
```

(continues on next page)

(continued from previous page)

```
'detector.names', 'verbosity', 'sampler.allExist', 'sampler.freeAll',
'sampler.raiseErrors', 'sampler.skipPrecheck', 'serpentVersion', 'xs.getInfXS',
'xs.getB1XS', 'xs.reshapeScatter', 'xs.variableGroups', 'xs.variableExtras'])
```

The *serpentTools.settings.rc* object and various `xs.*` settings can be used to control the *ResultsReader*. Specifically, these settings can be used to store only specific pieces of information. Here, we will store the version of Serpent, various cross sections, eigenvalues, and burnup data:

```
>>> rc['xs.variableGroups'] = ['versions', 'xs', 'eig', 'burnup-coeff']
```

Furthermore, we instruct the read to no read critical spectrum cross sections:

```
>>> rc['xs.getB1XS'] = False

>>> resFilt = serpentTools.readDataFile(resFile)
>>> resFilt.metadata.keys()
dict_keys(['version', 'compileDate', 'debug', 'title', 'confidentialData',
'inputFileName', 'workingDirectory', 'hostname', 'cpuType', 'cpuMhz',
'startDate', 'completeDate'])
>>> resFilt.resdata.keys()
dict_keys(['burnMaterials', 'burnMode', 'burnStep', 'burnup', 'burnDays',
'nubar', 'anaKeff', 'impKeff', 'colKeff', 'absKeff', 'absKinf', 'geomAlbedo'])
>>> univ0Filt = resFilt.getUniv('0', burnup=0.0, index=0, timeDays=0.0)
>>> univ0Filt.infExp.keys()
dict_keys(['infCapt', 'infAbs', 'infFiss', 'infNsf', 'infNubar', 'infKappa',
'infInvv', 'infScatt0', 'infScatt1', 'infScatt2', 'infScatt3', 'infScatt4',
'infScatt5', 'infScatt6', 'infScatt7', 'infTranspxs', 'infDiffcoef',
'infRabsxs', 'infRemxs', 'infChit', 'infChip', 'infChid', 'infS0', 'infS1',
'infS2', 'infS3', 'infS4', 'infS5', 'infS6', 'infS7'])
>>> univ0Filt.b1Exp
{}
```

4.7.6 Conclusion

The *ResultsReader* is capable of reading and storing all the data from the SERPENT `_res.m` file. Upon reading, the reader creates custom *HomogUniv* objects that are responsible for storing the universe related data. In addition, *metadata* and *resdata* are stored on the reader. These objects also have a handy *getUniv()* method for quick analysis of results corresponding to a specific universe and time point. Use of the *serpentTools.settings.rc* settings control object allows increased control over the data selected from the output file.

4.7.7 References

1. J. Leppanen, 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

4.8 Micro cross section reader

4.8.1 Basic Operation

This notebook demonstrates the capabilities of the serpentTools in regards to reading group micro cross-section files. SERPENT [1] produces a micro depletion file, containing independent and cumulative fission yields as well as group cross-sections for the isotopes and reactions defined by the user. The *MicroXSReader* is capable of reading this file, and storing the data directly on the reader. The *MicroXSReader* has two methods to retrieve the data and ease the analysis. Note: in order to obtain the micro depletion files, the user must set the `mdep` card in the input file.

Note: The preferred way to read your own output files is with the *serpentTools.read()* function. The `serpentTools.readDataFile()` function is used here to make it easier to reproduce the examples

---

```
>>> import serpentTools
>>> mdxFile = 'ref_mdx0.m'
>>> mdx = serpentTools.readDataFile(mdxFile)
```

The fission yields read in from the file are stored in the *nfy* dictionary, where the keys represent a specific (parent, energy) pair and the corresponding values is a dictionary with fission products ids and corresponding fission yield values.

```
>>> # All the (parent, energy) pairs can be obtained by using '.keys()'
>>> pairs = mdx.nfy.keys()
>>> list(pairs)[0:5] # list only the first five pairs
[(902270, 2.53e-08),
 (902280, 2.53e-08),
 (902280, 0.5),
 (902280, 14.0),
 (902290, 2.53e-08)]
```

Each pair represents the isotope undergoing fission and the impending neutron energy in MeV.

```
>>> pair = list(pairs)[0] # obtain the first (isotope, energy) pair
>>> print('Isotope= {: 1.0f}'.format(pair[0]))
Isotope=  902270
>>> print('Energy= {} MeV'.format(pair[1]))
Energy= 2.53e-08 MeV
```

The results for each pair are dictionaries that contain three fields:

1. `fissProd` list of fission products ids
2. `indYield` corresponding list of independent fission yields
3. `cumYield` corresponding list of cumulative fission yields

```
>>> # Obtain the keys in the nfy dictionary
>>> mdx.nfy[pair].keys()
```

(continues on next page)

(continued from previous page)

```
dict_keys(['fissProd', 'indYield', 'cumYield'])
>>> # Print only the five first fission products
>>> print(mdx.nfy[pair]['fissProd'][0:5])
[ 250660.  250670.  250680.  260660.  260670.]
>>> # Print only the five first fission independent yields
>>> print(mdx.nfy[pair]['indYield'][0:5])
[  6.97001000e-13   1.35000000e-13   1.01000000e-14   2.57000000e-10
1.13000000e-10]
>>> # Print only the five first fission cumulative yields
>>> print(mdx.nfy[pair]['cumYield'][0:5])
[  6.97001000e-13   1.35000000e-13   1.01000000e-14   2.58000000e-10
1.13000000e-10]
```

Fluxes ratios and uncertainties are stored in the *fluxRatio* and *fluxUnc* dictionaries, where the keys represent a specific universe and the corresponding values are group fluxes values.

```
>>> # obtain the universes
>>> print(mdx.fluxRatio.keys())
dict_keys(['0'])
```

Cross sections and their uncertainties are stored in the *xsVal* and *xsVal* dictionaries, where the keys represent a specific universe and the corresponding values are dictionaries. The keys within the nested dictionary describe the isotope, reaction and special flag:

```
>>> print(mdx.xsVal['0'].keys())
dict_keys([(10010, 102, 0), (982490, 18, 0), (982510, 102, 0), (982510, 16, 0),
(982510, 17, 0), (982510, 18, 0), (40090, 107, 0)])
```

Each key has three entries (isotope, reaction, flag)

1. `isotope` ID of the isotope (ZZAAA0/1), int or float
2. `reaction` MT e.g., 102 representing (n,gamma)
3. `flag` special flag to describe isomeric state or fission yield distribution number

For each such key (isotope, reaction, flag) the `xsVal` and `xsVal` store the group-wise flux values and uncertainties respectively.

```
>>> val = mdx.xsVal['0']
>>> unc = mdx.xsUnc['0']
>>> # Print flux values
>>> print(val[(10010, 102, 0)])
[  3.09753000e-05   3.33901000e-05   3.57054000e-05   3.70926000e-05
3.61049000e-05   3.39464000e-05   3.39767000e-05   3.98315000e-05
5.38962000e-05   7.96923000e-05   1.18509000e-04   1.73915000e-04
2.54571000e-04   3.38540000e-04   4.52415000e-04   5.98190000e-04
7.69483000e-04   1.04855000e-03   1.31149000e-03   1.67790000e-03
2.15195000e-03   2.70125000e-03   3.44635000e-03   5.04611000e-03]
>>> # Print flux uncertainties
>>> print(unc[(10010, 102, 0)])
[  1.10000000e-04   2.00000000e-05   1.00000000e-05   0.00000000e+00
0.00000000e+00   0.00000000e+00   0.00000000e+00   1.00000000e-05
1.00000000e-05   2.00000000e-05   2.00000000e-05   2.00000000e-05
2.00000000e-05   1.00000000e-05   1.00000000e-05   2.00000000e-05
2.00000000e-05   3.00000000e-05   2.00000000e-05   3.00000000e-05
4.00000000e-05   5.00000000e-05   1.70000000e-04   6.90000000e-04]
```

4.8.2 Data Retrieval

The *MicroXSReader* object has two `get` methods: 1. *getFY()* method obtains the independent and cumulative fission yields for a specific parent (ZZAAA0/1), daughter (ZZAAA0/1), neutron energy (MeV). If no parent or daaughter is found, the method raises an exception. The method also has a special flag that indicates whether the user wants to obtain the value corresponding to the nearest energy. 2. *getXS()* method to obtain the group-wise cross-sections for a specific universe, isotope and reaction.

```
>>> indYield, cumYield = mdx.getFY(parent=922350, energy=2.53e-08, daughter=541350 )
>>> print('Independent yield = {}'.format(indYield))
Independent yield = 0.000785125
>>> print('Cumulative yield = {}'.format(cumYield))
Cumulative yield = 0.065385
```

By default, the method includes a flag that allows to obtain the values for the closest energy defined by the user.

```
>>> indYield, cumYield = mdx.getFY(parent=922350, energy=1e-06, daughter=541350 )
>>> print('Independent yield = {}'.format(indYield))
Independent yield = 0.000785125
>>> print('Cumulative yield = {}'.format(cumYield))
Cumulative yield = 0.065385
```

The user can set this boolean flag to False if only the values at existing energies are of interest.

```
>>> indYield, cumYield = mdx.getFY(parent=922350, energy=2.53e-08, daughter=541350,␣
↪flagEnergy=False )
```

getXS() method is used to obtain the group cross-sections for a specific universe, isotope and reaction. The method returns the values and uncertainties.

```
>>> # Obtain the group cross-sections
>>> vals, unc = mdx.getXS(universe='0', isotope=10010, reaction=102)
>>> # Print group flux values
>>> print(vals)
[  3.09753000e-05   3.33901000e-05   3.57054000e-05   3.70926000e-05
3.61049000e-05   3.39464000e-05   3.39767000e-05   3.98315000e-05
5.38962000e-05   7.96923000e-05   1.18509000e-04   1.73915000e-04
2.54571000e-04   3.38540000e-04   4.52415000e-04   5.98190000e-04
7.69483000e-04   1.04855000e-03   1.31149000e-03   1.67790000e-03
2.15195000e-03   2.70125000e-03   3.44635000e-03   5.04611000e-03]
>>> # Print group flux uncertainties values
>>> print(unc)
[  1.10000000e-04   2.00000000e-05   1.00000000e-05   0.00000000e+00
0.00000000e+00   0.00000000e+00   0.00000000e+00   1.00000000e-05
1.00000000e-05   2.00000000e-05   2.00000000e-05   2.00000000e-05
2.00000000e-05   1.00000000e-05   1.00000000e-05   2.00000000e-05
2.00000000e-05   3.00000000e-05   2.00000000e-05   3.00000000e-05
4.00000000e-05   5.00000000e-05   1.70000000e-04   6.90000000e-04]
```

The method includes a special flag `isomeric`, which is set to zero by default. The special flag either describes the isomeric state or fission yield distribution number.

```
>>> # Example of how to use the isomeric flag
>>> vals, unc = mdx.getXS(universe='0', isotope=10010, reaction=102, isomeric=0)
```

If the universe exist, but the isotope or reaction do not exist, the method raises an error.

4.8.3 Settings

The *MicroXSReader* also has a collection of *serpentTools.settings.rc* 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
>>> rc['microxs.getFY'] = False # True/False only
>>> rc['microxs.getXS'] = True # True/False only
>>> rc['microxs.getFlx'] = True # True/False only
```

- *microxs.getFY*: True or False, store fission yields
- *microxs.getXS*: True or False, store group cross-sections and uncertainties
- *microxs.getFlx*: True or False, store flux ratios and uncertainties

```
>>> mdx = serpentTools.readDataFile(mdxFile)
>>> # fission yields are not stored on the reader
>>> mdx.nfy.keys()
dict_keys([])
```

4.8.4 Conclusion

The *MicroXSReader* is capable of reading and storing all the data from the SERPENT micro depletion file. Fission yields, cross-sections and flux ratios are stored on the reader. The reader also includes two methods *getFY()* and *getXS()* to retrieve the data. Use of *serpentTools.settings.rc* settings control object allows increased control over the data selected from the output file.

4.8.5 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

4.9 Depletion Matrix Reader

The `serpentTools` package supports reading depletion matrix files, generated when `set depmtx 1` is added to the input file. As of `SERPENT` 2.1.30, these files contain

1. The length of time for a depletion interval
2. Vector of initial concentrations for all isotopes present in the depletion problem
3. ZAI vector
4. Depletion matrix governing destruction and transmutation of isotopes
5. Vector of final concentrations following one depletion event

Files such as this are present for each burnable material tracked by `SERPENT` and at each time step in the problem.

This document will demonstrate the *DepmtxReader*, designed to store such data.

Note: The depletion matrices can be very large for most problems, ~1000 x 1000 elements. For this reason, the *DepmtxReader* can store matrices in Compressed Sparse Column or full *numpy* arrays. The reader will use the sparse format if *scipy* is installed unless explicitely told to use dense arrays.

4.9.1 Basic Operation

Note: The preferred way to read your own output files is with the *serpentTools.read()* function. The `serpentTools.readDataFile()` function is used here to make it easier to reproduce the examples

```
>>> import serpentTools
>>> reader = serpentTools.readDataFile('depmtx_ref.m')
>>> reader
<serpentTools.parsers.depmatrix.DepmtxReader at 0x7f0b1a9702b0>
```

We not have access to all the data present in the fire directly on the reader.

```
>>> reader.n0
array([1.17344222e-07, 6.10756908e-12, 7.48053806e-13, 7.52406757e-16,
       1.66113020e-34, 1.67580185e-09, 1.19223790e-36, 1.89040622e-26,
       5.09195054e-16, 7.91142112e-34, 1.68989876e-22, 6.92676695e-12,
       7.52406345e-16, 8.52076751e-13, 4.52429540e-02, 1.71307881e-12,
       1.86228871e-51, 2.32287315e-50, 1.15352152e-55, 7.72524686e-50,
       5.74084741e-44, 1.55414063e-42, 3.10757266e-40, 9.12566461e-40,
       6.82216144e-39, 9.71825616e-56, 1.59237444e-51, 1.14764875e-46,
       1.15203415e-43, 5.66072799e-41, 4.49411601e-34, 8.99210202e-31,
       8.65694179e-29, 5.96910982e-28, 1.06642058e-26, 9.10883647e-27,
       7.56006632e-36, 6.08157358e-33, 7.93562601e-40, 1.67857401e-29,
       2.76995718e-26, 2.42939173e-30, 6.93658246e-27, 3.21960435e-20,
       4.14863808e-17, 6.02145579e-16, 3.68254657e-15, 2.25927183e-15,
       2.85992932e-15, 5.34540710e-28, 2.34532631e-25, 1.36140065e-17,
       4.17935379e-16, 4.61527247e-15, 2.15346589e-15, 2.90307762e-15,
       4.90358169e-16, 3.62499544e-13, 4.61691784e-05, 2.96919439e-04,
       4.13730091e-04, 3.14746134e-04, 4.98296713e-04, 4.37637914e-04,
       3.84679634e-17, 6.87038906e-14, 4.29307714e-06, 5.62156587e-04,
       2.98288610e-08, 1.45634092e-09, 2.05487374e-02, 2.10836706e-07,
       9.84180195e-12, 8.05226656e-16], dtype=float128)
```

This specific input file did not include fission yield libraries and thus only tracks 74 isotopes, rather than 1000+, through depletion. This was intentionally done to reduce the size of files tracked in this project.

Number densities and entries in the depletion matrix are stored using the `numpy.longfloat` data type to preserve the precision present in the output files.

```
>>> reader.zai
array([    -1,  10010,  10020,  10030,  20030,  20040,  30060,  30070,
        40090,  50100,  50110,  60120,  70140,  70150,  80160,  80170,
        561380, 561400, 581380, 581390, 581400, 581410, 581420, 581430,
        581440, 591410, 591420, 591430, 601420, 601430, 601440, 601450,
        601460, 601470, 601480, 601500, 611470, 611480, 611481, 611490,
        611510, 621470, 621480, 621490, 621500, 621510, 621520, 621530,
        621540, 631510, 631520, 631530, 631540, 631550, 631560, 631570,
        641520, 641530, 641540, 641550, 641560, 641570, 641580, 641600,
```

(continues on next page)

(continued from previous page)

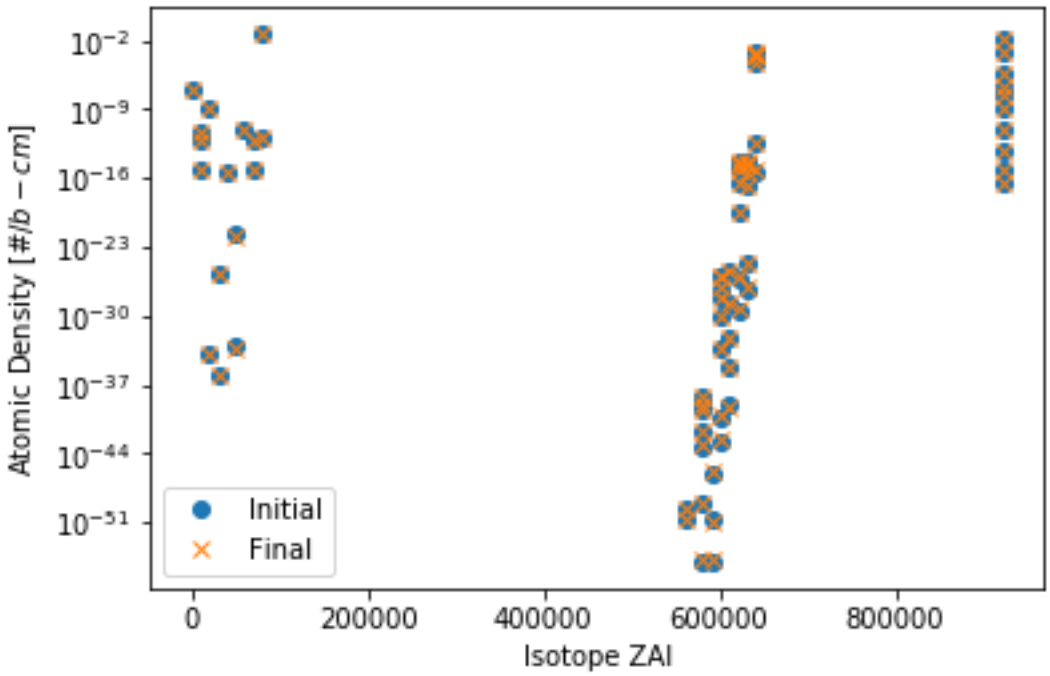
```
        922320, 922330, 922340, 922350, 922360, 922370, 922380, 922390,
        922400, 922410])
```

One can easily check if the depletion matrix is stored in a sparse or dense structure using the `sparse` attribute:

```
>>> reader.sparse
True
>>> reader.depmtx
<74x74 sparse matrix of type ',class 'numpy.float128'>'
   with 633 stored elements in Compressed Sparse Column format>
```

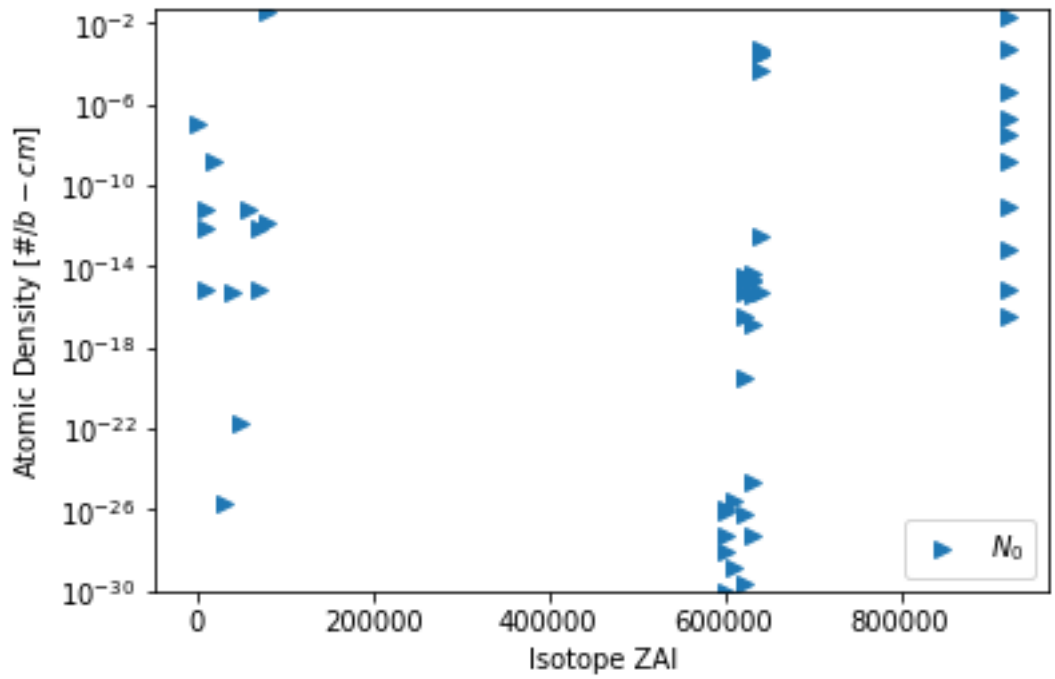
A simple plot method can be used to plot initial concentrations, final concentrations, or both:

```
>>> reader.plotDensity()
```

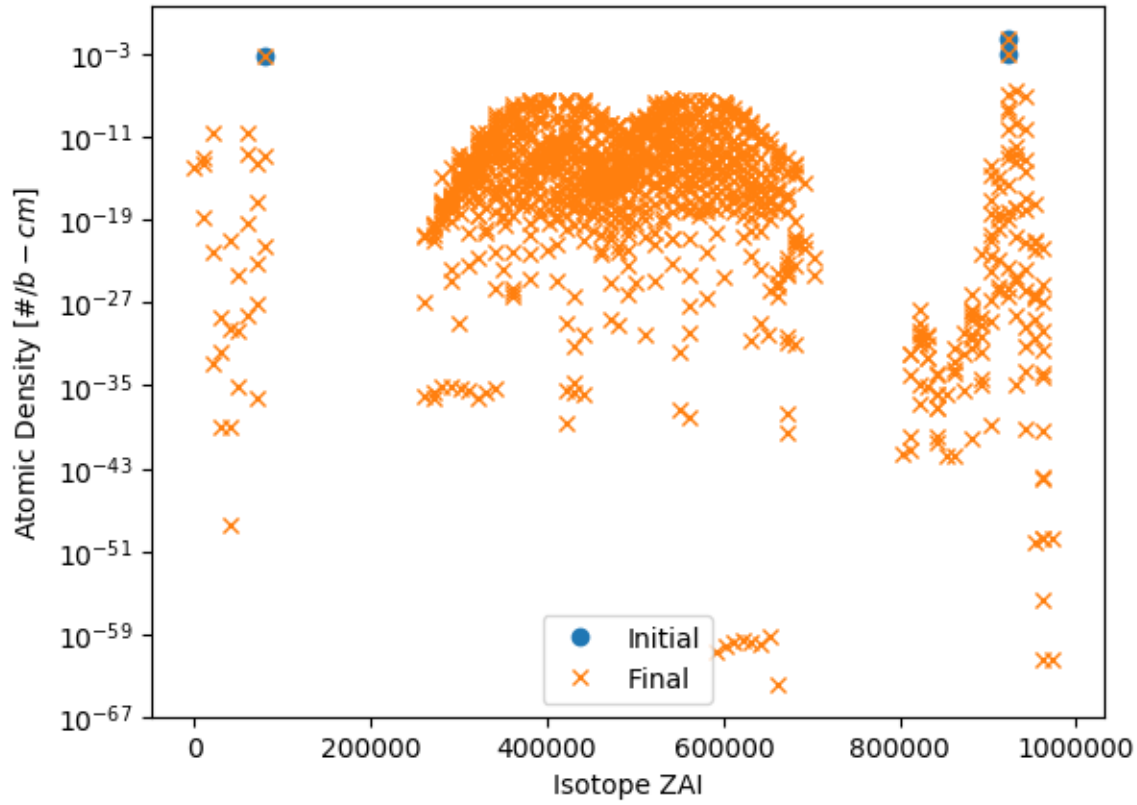


Some options can be passed to alter the formatting of the plot:

```
>>> reader.plotDensity(
      what='n0',  # plot only initial concentration
      markers='>',  # marker for scatter plot
      labels='$N_0$'  # label for each plotted entry
      ylim=1E-30,  # set the lower y-axis limit
   )
```



We can see that there is not a lot of change in the isotopic concentration in this depletion step. Furthermore, the classical fission yield distributions are not present due to the lack of fission yield data. Using a more complete, and typical data set, one can view the distribution of fission products more clearly, demonstrated in the below plot.



4.10 Coefficient file to nodal diffusion cross sections

A recent feature of SERPENT is the ability to performing branching calculations using the automated burnup sequence. `serpentTools` can read these coefficient files using the *BranchingReader* This automated burnup sequence is ideal for generating group constant data for nodal diffusion codes, that often include some multi-physics features, criticality searches, or other control mechanisms. A criticality search could be performed by tweaking the boron concentration in the coolant or adjusting control rod insertions. Similarly, some codes may include coupled TH analysis to convert power profiles to temperature profiles and adjust cross sections accordingly. Each code has a unique flavor for utilizing a set of group constants across these perturbations, and this notebook will demonstrate using the *BranchCollector* to gather and write a simple set of cross sections.

```
>>> import numpy
>>> import serpentTools
>>> from serpentTools.xs import BranchCollector
>>> # use serpentTools.read for everything except following along with examples
>>> coe = serpentTools.readDataFile('demo.coe')
```

This specific input file contained two perturbations: boron concentration and fuel temperature. Boron concentration had three branches: `nom` with no boron, then `B1000` and `B750`, with 1000 and 750 ppm boron in coolant. Fuel temperature had a nominal branch at 900 K, with 1200 and 600 K perturbations as well. These can be confirmed by observing the *branches* dictionary on the *BranchingReader*.

```
>>> list(coe.branches.keys())
[('nom', 'nom'),
 ('B750', 'nom'),
 ('B1000', 'nom'),
 ('nom', 'FT1200'),
('B750', 'FT1200'),
 ('B1000', 'FT1200'),
 ('nom', 'FT600'),
 ('B750',
'FT600'),
 ('B1000', 'FT600')]
```

Cross sections are spread out through this *BranchingReader* across branches, burnup, and universes. The job of the *BranchCollector* is to place that data into mutli-dimensional matrices that represent the perturbations chosen by the user. A single group constant, say total cross section, has unique values for each universe, at each burnup point, for each perturbed state, and each energy group. Such a matrix would then contain five dimensions for this case.

First, we create the *BranchCollector* from the *BranchingReader* and instruct the reader what perturbations are present in the file. The ordering is not important at this point, as it can be changed later.

```
>>> collector = BranchCollector(coe)
>>> collector.collect(('BOR', 'TFU'))
```

Now we can inspect the perturbation states, `states` found by the collector.

```
>>> collector.states
(('B1000', 'B750', 'nom'), ('FT1200', 'FT600', 'nom'))
```

The group constants are stored in the `xsTables` dictionary. Here we select the total cross section, `infTot` for further exploration.

```
>>> list(collector.xsTables.keys())
['infTot', 'infFiss', 'infS0', 'infS1',
 'infDiffcoef', 'b1Tot', 'b1Fiss', 'b1S0',
```

(continues on next page)

(continued from previous page)

```
 'b1S1', 'b1Diffcoef']
>>> infT = collector.xsTables['infTot']
>>> infT.shape
(5, 3, 3, 3, 2)
```

Five dimensions as mentioned above. But how are they ordered? Inspecting the `axis` attribute tells us that the dimensions are universe, boron concentration, fuel temperature, burnup, and energy group.

```
>>> collector.axis
('Universe', 'BOR', 'TFU', 'Burnup', 'Group')
```

The ordering of each of these dimensions is found by examining the `univIndex`, `states` and `burnups` attributes.

```
>>> collector.univIndex
(0, 10, 20, 30, 40)
>>> collector.states
(('B1000', 'B750', 'nom'), ('FT1200', 'FT600', 'nom'))
>>> collector.burnups
array([ 0.,  1., 10.])
```

For example, if we wanted the total cross section for universe 10, at 1000 ppm boron, nominal fuel temperature, and 10 MWd/kgU burnup, we would request

```
>>> infT[1, 0, 2, 2]
array([0.324746, 0.864346])
```

For this example, the scattering matrices were not reshaped from vectors to matrices and we would observe slightly different behavior in the `'Group'` dimension.

```
>>> collector.xsTables['infS1'].shape
(5, 3, 3, 3, 4)
```

Four items in the last axis as the vectorized matrix represents fast to fast, fast to thermal, thermal to fast, and thermal to thermal scattering.

```
>>> collector.xsTables['infS1'][1, 0, 2, 2]
array([0.087809  , 0.00023068, 0.00073939, 0.123981  ])
```

Many nodal diffusion codes request group constants on a per universe basis, or per assembly type. As we saw above, the first dimension of the `xsTables` matrices corresponds to universe. One can view group constants for specific universes with the `universes` dictionary.

```
>>> collector.universes
{"0": <serpentTools.BranchedUniv at 0x7fb62f749a98>, 10:
<serpentTools.BranchedUniv at 0x7fb62f731b88>, 20:
<serpentTools.BranchedUniv at 0x7fb62f749e08>, 30:
<serpentTools.BranchedUniv at 0x7fb62f749e58>, 40:
<serpentTools.BranchedUniv at 0x7fb62f749ea8>}
>>> u0 = collector.universes["0"]
```

These *`BranchedUniv`* objects store views into the underlying collectors *`xsTables`* data corresponding to a single universe. The structuring is identical to that of the collector, with the first axis removed.

```
>>> u0.perturbations
('BOR', 'TFU')
>>> u0.axis
```

(continues on next page)

(continued from previous page)

```
('BOR', 'TFU', 'Burnup', 'Group')
>>> u0.states
(('B1000', 'B750', 'nom'), ('FT1200', 'FT600', 'nom'))
```

The contents of the *xsTables* dictionary are `numpy.array` views into the data stored on the collector.

```
>>> list(u0.xsTables.keys())
['infTot', 'infFiss', 'infS0', 'infS1',
 'infDiffcoef', 'b1Tot', 'b1Fiss', 'b1S0',
 'b1S1', 'b1Diffcoef']
>>> u0Tot = u0.xsTables['infTot']
>>> u0Tot.shape
(3, 3, 3, 2)
>>> u0Tot
array([[[[0.313696, 0.544846],
         [0.311024, 0.617734],
         [0.313348, 0.614651]],

        [[0.313338, 0.54515 ],
         [0.310842, 0.618286],
         [0.31299 , 0.614391]],

         ...

        [[0.210873, 0.223528],
         [0.208646, 0.      ],
         [0.206532, 0.      ]]]])
```

4.10.1 Changing perturbation values

The values of `states` and `perturbations` can be easily modified, so long as the structures are preserved. For example, as the current `states` are string values, and of equal perturbations (three boron concentrations, three fuel temperatures), we can set the `states` to be a single 2x3 array

```
>>> collector.states = numpy.array([
...     [1000, 750, 0],
...     [1200, 600, 900]],
...     dtype=float)
>>> collector.states
array([[1000.,  750.,    0.],
       [1200.,  600.,  900.]])
```

Some error checking is performed to make sure the passed perturbations match the structure of the underlying data. Here, we attempt to pass the wrong number of fuel temperature perturbations.

```
>>> try:
...     collector.states = numpy.array([
...         [1000, 750, 0],
...         [1200, 600],  # wrong
...     ])
>>> except ValueError as ve:
...     print(str(ve))

Current number of perturbations for state TFU is 3, not 2
```

If the specific perturbations were not known when creating the collector, the value of `perturbations` can also be changed, with similar error checking.

```
>>> collector.perturbations = ['boron conc', 'fuel temperature']
>>> collector.perturbations
['boron conc', 'fuel temperature']
>>> try:
...     collector.perturbations = ['boron', 'fuel', 'ctrl']  # wrong
>>> except ValueError as ve:
...     print(str(ve))
Current number of perturbations is 2, not 3
```

4.10.2 Example nodal diffusion writer

As each nodal diffusion code has it's own required data structure, creating a general writer is a difficult task. The intent with the *BranchCollector* is to provide a framework where the data is readily available, and such a writer can be created with ease. Here, an example writer is demonstrated, one that writes each cross section. The writer first writes a table of the perturbations at the top of the input file, showing the ordering and values of the perturbations. Options are also provided for controlling formatting.

The full file is available for download: nodal_writer.py

```
>>> from nodal_writer import Writer
>>> print(Writer.__doc__.strip())
Class for writing an example cross section file.

Parameters
----------
collector: Collector
        Object that read the branching file and stored
the cross sections
        along the perturbation vector
    xsPerLine: int
Number of cross sections / group constants to write per line
    floatFmt: str
Formattable string used when writing floating point values
    strFmt: str
Formattable string used when writing the names of the perturbations
xsRemap: None or dict
        Dictionary used to find a replacement name for
cross sections when
        writing.  Between each cross section block, the
name of cross
        section and group will be written as ``# {name} group
{g}``.
        When ``xsRemap`` is ``None``, the names are ``mixedCase`` as
they appear in ``HomogUniv`` objects, e.g.  ``'infTot'``,
``'diffCoeff'``, etc. If ``xsRemap`` is a dictionary, it can
        be used to
write a different name. Passing ``{'infTot': 'Total
        cross section'}``
would write ``'Total cross seciton'``
        instead of ``'infTot'``, but all
other names would be unchanged.

>>> writer = Writer(collector)
```

(continues on next page)

(continued from previous page)

```
>>> print(writer.write.__doc__.strip())
Write the contents of a single universe

Parameters
----------
universe: int or key
            Key of universe that exists in
``self.collector``. Typically
            integer values of homogenized
universes from coefficient file
        stream: None or str or writeable
If ``None``, return a string containing what would have been
written to file. If a string, then write to this file. Otherwise,
ensure that the object has a ``write`` method and write to this
object
        mode: {'a', 'w'}
            Write or append to file. Only
needed if stream is a string

>>> # write to a file "in memory"
>>> out = writer.write(0)
>>> print(out[:1000])
# Cross sections for universe 0
boron conc           1.00000000E+03
7.50000000E+02 0.00000000E+00
fuel temperature     1.20000000E+03
6.00000000E+02 9.00000000E+02
Burnup [MWd/kgU]     0.00000000E+00
1.00000000E+00 1.00000000E+01
# infTot group 1
 3.13696000E-01 3.11024000E-01
3.13348000E-01 3.13338000E-01
 3.10842000E-01 3.12990000E-01 3.16730000E-01
3.13987000E-01
 3.16273000E-01 3.13772000E-01 3.11335000E-01 3.13311000E-01
3.13437000E-01 3.10967000E-01 3.13160000E-01 3.16688000E-01
 3.14245000E-01
3.16392000E-01 2.08020000E-01 2.05774000E-01
 2.03646000E-01 2.07432000E-01
2.05326000E-01 2.03533000E-01
 2.10873000E-01 2.08646000E-01 2.06532000E-01
#
infTot group 2
 5.44846000E-01 6.17734000E-01 6.14651000E-01 5.45150000E-01
6.18286000E-01 6.14391000E-01 5.48305000E-01 6.21804000E-01
 6.18120000E-01
5.41505000E-01 6.09197000E-01 6.08837000E-01
 5.42373000E-01 6.09192000E-01
6.08756000E-01 5.45294000E-01
 6.12767000E-01 6.12985000E-01 2.28908000E-01
1.07070000E-01
 0.00000000E+00 3.1
```

CHAPTER

FIVE

FILE-PARSING API

This page contains links to the full documentation for the readers used by `serpentTools`.

5.1 Functions

The core of `serpentTools` is the *`read()`* function. This function attempts to determine the file type based on the extension and use the correct reader to parse the file. Below are links to primary reader functions that accept a file path argument and will process the file accordingly.

`read`	Simple entry point to read a file and obtain the processed reader.
`readDepmtx`	Simple entry point to obtain data from depletion matrix files

5.1.1 serpentTools.read

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

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

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
branch	`BranchingReader`
bumat	`BumatReader`
dep	`DepletionReader`
det	`DetectorReader`
fission	`FissionMatrixReader`
history	`HistoryReader`
microxs	`MdxReader`
results	`ResultsReader`
sensitivity	`SensitivityReader`
xsplot	`XSPlotReader`

Returns Correct subclass corresponding to the file type

Return type serpentTools.objects.base.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.

5.1.2 serpentTools.readDepmtx

`serpentTools.`**`readDepmtx`**(*filePath*, *sparse=True*)

Simple entry point to obtain data from depletion matrix files

Parameters

- **filePath** (*str*) – Path of file to be read
- **sparse** (*bool*) – If this is `True`, attempt to construct a sparse depletion matrix, rather than a dense matrix. Requires `scipy`. If `scipy` is not installed, a full matrix will be returned

Returns

- **dt** (*float*) – Length of depletion interval
- **n0** (`numpy.ndarray`) – 1D vector of initial isotopics. Ordered according to `zai` vector
- **zai** (`numpy.ndarray`) – 1D vector of `ZZAAAI` isotope identifiers
- **A** (`numpy.ndarray` or `scipy.sparse.csc_matrix`) – Depletion matrix governing decay/production of isotopics. Will be a sparse matrix if `sparse` was passed as `True` and `scipy` is installed.
- **n1** (`numpy.ndarray`) – 1D vector of isotopics after depletion event.

5.2 Classes

The *read()* function relies heavily on the following parsers which can be created and used outside of the function. These classes are also the returned types for *read()*.

BumatReader	Parser responsible for reading and working with burned material files.
BranchingReader	Parser responsible for reading and working with automated branching files.
DepletionReader	Parser responsible for reading and working with depletion files.
DepmtxReader	Reader for processing depletion matrix files
DetectorReader	Parser responsible for reading and working with detector files.
HistoryReader	Class responsible for reading history files

Continued on next page

Table 2 – continued from previous page

MicroXSReader	Parser responsible for reading and working with micro-xs (mdx) files.
ResultsReader	Parser responsible for reading and working with result files.
SensitivityReader	Class for reading sensitivity files
XSPlotReader	Parser responsible for reading and working with xsplot output files.

5.2.1 serpentTools.BumatReader

class serpentTools.**BumatReader**(*filePath*)

Parser responsible for reading and working with burned material files.

..note:

```
This is experimental and will be subject to change depending
upon the implementation of GH Issue #12
```

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

materials

Dictionary of materials with keys as names and values being a dictionary of parameters. **The storage of materials is subject to change**

Type dict

burnup

Burnup [MWd/kgU] for this file

Type float

days

Burnup [days] for this file

Type float

5.2.2 serpentTools.BranchingReader

class serpentTools.**BranchingReader**(*filePath*)

Parser responsible for reading and working with automated branching files.

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

branches

Dictionary of branch names and their corresponding *BranchContainer* objects

Type dict

property hasUncs

boolean if uncertainties are present in the file

iterBranches()

Iterate over branches yielding paired branch IDs and containers

5.2.3 serpentTools.DepletionReader

class serpentTools.**DepletionReader**(*filePath*)

Parser responsible for reading and working with depletion files.

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

materials

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

Type dict

metadata

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

Type dict

settings

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

Type dict

__getitem__(*name*)

Retrieve a material from *materials*.

compareMaterials(*other*, *lower=0*, *upper=10*, *sigma=2*)

Return the result of comparing all materials on two readers

Parameters

- **other** (*DepletionReader*) – Reader to compare against
- **lower** (*float or int*) – Lower limit for relative tolerances in percent Differences below this will be considered allowable
- **upper** (*float or int*) – Upper limit for relative tolerances in percent. Differences above this will be considered failure and errors messages will be raised
- **sigma** (*int*) – Size of confidence interval to apply to quantities with uncertainties. Quantities that do not have overlapping confidence intervals will fail

Returns True if all materials agree to the given tolerances

Return type bool

Raises **TypeError** – If other is not of the same class as this class nor a subclass of this class

compareMetadata(*other*, *lower=0*, *upper=10*, *sigma=2*)

Return the result of comparing metadata on two readers

Parameters

- **other** (*DepletionReader*) – Object to compare against
- **lower** (*float or int*) – Lower limit for relative tolerances in percent Differences below this will be considered allowable
- **upper** (*float or int*) – Upper limit for relative tolerances in percent. Differences above this will be considered failure and errors messages will be raised
- **header** (*bool*) – Print/log an info message about this comparison.

Returns True if the metadata agree within the given tolerances

Return type bool

Raises **TypeError** – If `other` is not of the same class as this class nor a subclass of this class

saveAsMatlab(*fileP*, *reconvert=True*, *metadata=None*, *append=True*, *format='5'*, *longNames=True*, *compress=True*, *oned='row'*)

Write a binary MATLAB file from the contents of this reader

Deprecated since version 0.7.0: Use *toMatlab()*

Parameters

- **fileP** (*str or file-like object*) – Name of the file to write
- **reconvert** (*bool*) – If this evaluates to true, convert values back into their original form as they appear in the output file, e.g. `MAT_TOTAL_ING_TOX`. Otherwise, maintain the `mixedCase` style, `total_ingTox`.
- **metadata** (*bool or str or list of strings*) – If this evaluates to true, then write all metadata to the file as well.
- **append** (*bool*) – If true and a file exists under `output`, append to that file. Otherwise the file will be overwritten
- **format** (*{'5', '4'}*) – Format of file to write. `'5'` for MATLAB 5 to 7.2, `'4'` for MATLAB 4
- **longNames** (*bool*) – If true, allow variable names to reach 63 characters, which works with MATLAB 7.6+. Otherwise, maximum length is 31 characters
- **compress** (*bool*) – If true, compress matrices on write
- **oned** (*{'row', 'col'}:*) – Write one-dimensional arrays as row vectors if `oned=='row'` (default), or column vectors

:raises ImportError:: If *scipy* is not installed

See also:

`scipy.io.savemat()`

toMatlab(*fileP*, *reconvert=True*, *append=True*, *format='5'*, *longNames=True*, *compress=True*, *oned='row'*)

Write a binary MATLAB file from the contents of this object

Parameters

- **fileP** (*str or file-like object*) – Name of the file to write. `.mat` extension is not needed if `append==True`
- **reconvert** (*bool*) – If this evaluates to true, convert values back into their original form as they appear in the output file.
- **append** (*bool*) – If true and a file exists under `output`, append to that file. Otherwise the file will be overwritten
- **format** (*{'5', '4'}*) – Format of file to write. `'5'` for MATLAB 5 to 7.2, `'4'` for MATLAB 4
- **longNames** (*bool*) – If true, allow variable names to reach 63 characters, which works with MATLAB 7.6+. Otherwise, maximum length is 31 characters
- **compress** (*bool*) – If true, compress matrices on write
- **oned** (*{'row', 'col'}*) – Write one-dimensional arrays as row vectors if `oned=='row'` (default), or column vectors

Raises **ImportError** – If *scipy* is not installed

See also:

- scipy.io.savemat()

5.2.4 serpentTools.DepmtxReader

class serpentTools.**DepmtxReader**(*filePath*, *sparse=None*)
Reader for processing depletion matrix files

Parameters

- **filePath** (*str*) – Path to file to be read
- **sparse** (bool or None) – Use *scipy* sparse matrices if True. If passed as None, only use sparse if *scipy* is installed

deltaT
Length of depletion interval

Type float

n0
Vector of original isotopics

Type numpy.ndarray

zai
Integer vector of isotope ZAI (zzaaai) identifiers

Type numpy.ndarray

depmtx
Depletion matrix for this material. Will be scipy.sparse.csc_matrix if *scipy* is installed and the sparse-engine option is either not specified (None) during *read()* or passed as True. Otherwise, will be a numpy.ndarray

Type numpy.ndarray or

n1
Vector for isotopics after depletion

Type numpy.ndarray

plotDensity(*what='both'*, *ax=None*, *logx=False*, *logy=True*, *loglog=None*, *legend=None*, *title=None*, *labels=None*, *markers=None*, *xlabel=None*, *ylabel=None*, *ylim=None*)
Plot initial, final, or both number densities.

Parameters

- **what** (*str*) – Concentrations to plot.
  1. 'both': plot initial and final
  2. 'n0' or 'initial': only initial
  3. 'n1' or 'final': only final
- **ax** (matplotlib.axes.Axes, optional) – Ax on which to plot the data. If not provided, create a new plot
- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.

- **legend** (*bool or str or None*) – Automatically label the plot. No legend will be made if a single item is plotted. Pass one of the following values to place the legend outside the plot: above, right
- **title** (*str*) – Title to apply to the figure.
- **labels** (*None or str or list of strings*) – Labels to apply to concentration plot(s). If given, must have length equal to the number of quantities plotted, e.g. plotting both concentrations and passing a single string for `labels` is not allowed
- **markers** (*None or str or list of strings*) – Markers to apply to each plot. Must be a valid `matplotlib` marker such as `'o'`, `'>'`, etc. If given, the number of markers given must equal the number of quantities to plot.
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylim** (*None or float or list of floats*) – If a single value is given, set the lower y-axis limit to this value. If two values are given, set the upper and lower y-axis limits to this value

Returns Ax on which the data was plotted.

Return type `matplotlib.axes.Axes`

toMatlab(*fileP*, *reconvert=True*, *append=True*, *format='5'*, *longNames=True*, *compress=True*, *oned='row'*)

Write a binary MATLAB file from the contents of this object

Parameters

- **fileP** (*str or file-like object*) – Name of the file to write. `.mat` extension is not needed if `append==True`
- **reconvert** (*bool*) – If this evaluates to true, convert values back into their original form as they appear in the output file.
- **append** (*bool*) – If true and a file exists under `output`, append to that file. Otherwise the file will be overwritten
- **format** (*{'5', '4'}*) – Format of file to write. `'5'` for MATLAB 5 to 7.2, `'4'` for MATLAB 4
- **longNames** (*bool*) – If true, allow variable names to reach 63 characters, which works with MATLAB 7.6+. Otherwise, maximum length is 31 characters
- **compress** (*bool*) – If true, compress matrices on write
- **oned** (*{'row', 'col'}*) – Write one-dimensional arrays as row vectors if `oned=='row'` (default), or column vectors

Raises **ImportError** – If *scipy* is not installed

See also:

- `scipy.io.savemat()`

5.2.5 serpentTools.DetectorReader

class serpentTools.**DetectorReader**(*filePath*)
Parser responsible for reading and working with detector files.

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

detectors
Dictionary where key, value pairs correspond to detector names and their respective Detector instances

Type dict

__getitem__(*name*)
Retrieve a detector from *detectors*

iterDets()
Yield name, detector pairs by iterating over *detectors*.

toMatlab(*fileP*, *reconvert=True*, *append=True*, *format='5'*, *longNames=True*, *compress=True*, *oned='row'*)
Write a binary MATLAB file from the contents of this object

Parameters

- **fileP** (*str or file-like object*) – Name of the file to write. .mat extension is not needed if append==True
- **reconvert** (*bool*) – If this evaluates to true, convert values back into their original form as they appear in the output file.
- **append** (*bool*) – If true and a file exists under output, append to that file. Otherwise the file will be overwritten
- **format** (*{'5', '4'}*) – Format of file to write. '5' for MATLAB 5 to 7.2, '4' for MATLAB 4
- **longNames** (*bool*) – If true, allow variable names to reach 63 characters, which works with MATLAB 7.6+. Otherwise, maximum length is 31 characters
- **compress** (*bool*) – If true, compress matrices on write
- **oned** (*{'row', 'col'}*) – Write one-dimensional arrays as row vectors if oned=='row' (default), or column vectors

Raises **ImportError** – If *scipy* is not installed

See also:

- scipy.io.savemat()

5.2.6 serpentTools.HistoryReader

class serpentTools.**HistoryReader**(*filePath*)
Class responsible for reading history files

Arrays can be accessed through either the arrays dictionary, or with his[key], where key is the name of an array in arrays.

Parameters **filePath** (*str*) – path pointing towards the file to be read

arrays
Dictionary of all the arrays produced in the file. These arrays do not have the index column that is presented in the file.

Type dict

numInactive
Number of inactive cycles used in this calculation

Type int

__contains__(*key*)
Return `True` if key is in *arrays*, otherwise `False`

__iter__()
Iterate over keys in *arrays*

__len__()
Return number of entries in *arrays*.

static ioConvertName(*name*)
Convert a variable name to `camelCase` for exporting.

static ioReconvertName(*name*)
Reconvert a variable name to `SERPENT_STYLE` for exporting

items()
Iterate over `(key, value)` pairs from *arrays*

toMatlab(*fileP*, *reconvert=True*, *append=True*, *format='5'*, *longNames=True*, *compress=True*, *oned='row'*)
Write a binary MATLAB file from the contents of this object

Parameters

- **fileP** (*str or file-like object*) – Name of the file to write. `.mat` extension is not needed if `append==True`
- **reconvert** (*bool*) – If this evaluates to true, convert values back into their original form as they appear in the output file.
- **append** (*bool*) – If true and a file exists under `output`, append to that file. Otherwise the file will be overwritten
- **format** (*{'5', '4'}*) – Format of file to write. `'5'` for MATLAB 5 to 7.2, `'4'` for MATLAB 4
- **longNames** (*bool*) – If true, allow variable names to reach 63 characters, which works with MATLAB 7.6+. Otherwise, maximum length is 31 characters
- **compress** (*bool*) – If true, compress matrices on write
- **oned** (*{'row', 'col'}*) – Write one-dimensional arrays as row vectors if `oned=='row'` (default), or column vectors

Raises **ImportError** – If *scipy* is not installed

See also:

- `scipy.io.savemat()`

5.2.7 serpentTools.MicroXSReader

class serpentTools.**MicroXSReader**(*filePath*)

Parser responsible for reading and working with micro-xs (mdx) files.

Parameters **filePath** (*str*) – path to the `*mdx[n].m` file

nfy

Nested dictionary with tuples (parent, energy) as keys. Parent is the isotope undergoing fission, e.g. 922350 and energy is the impending neutron energy causing fission in MeV. The values are nested dictionaries with the following structure:

```
"fissProd": list of fission product ZAI ids, e.g. [541350, 551350, ...]
"indYield": list of independent yields
"cumYield": list of cumulative yields
```

Type dict

fluxRatio

Dictionary with universes id as keys and the corresponding group-wise flux values. e.g., `fluxRatio['0'] = [9.91938E+14, 1.81729E+15]`

Type dict

fluxUnc

Dictionary with universes id as keys and the corresponding group-wise flux uncertainty values. e.g., `fluxRatio['0'] = [0.00023, 0.00042]`

Type dict

xsVal

Expected value on microscopic cross sections, sorted by universe then by isotope, reaction, and metastable flag. Nested dictionary with universes as keys, e.g. '0'. The values are nested dictionary with `MicroXSTuple` as keys (isotope, reaction, flag) and group xs as values. e.g., `(922350, 18, 0)`

Type dict

xsUnc

Uncertainties on microscopic cross sections, sorted by universe then by isotope, reaction, and metastable flag. Nested dictionary with universes as keys, e.g. '0'. The values are nested dictionary with `MicroXSTuple` as keys (isotope, reaction, flag) and group xs as values. e.g., `(922350, 18, 0)`

Type dict

Raises ***SerpentToolsException*** – No results exist in the file, or no results are collected

getFY(*parent*, *energy*, *daughter*, *flagEnergy=True*)

Return a specific fission yield given the parent ID, neutron energy and daughter ID

If the energy does not exist in the results, the fission yield corresponding to the closest energy is returned

Parameters

- **parent** (*int or float*) – ID of the parent undergoing fission
- **energy** (*float*) – neutron energy in MeV
- **daughter** (*int or float*) – ID of the fission product
- **flagEnergy** (*boolean*) – If set to true, the function will return the fission yield that matches to the closest energy given by the user

Returns

- **indYield** (*float*) – Independent fission yield
- **cumYield** (*float*) – Cumulative fission yield

:raises SerpentToolsException:: If energy is a negative number If parent or fission product are not found

getXS (*universe*, *isotope*, *reaction*, *isomeric=0*)

Return the group-wise micro cross-sections for a specific isotope, and reaction

Parameters

- **universe** (*string*) – universe ID, e.g., *0*
- **isotope** (*int or float*) – ID of the isotope (ZZAAA0/1)
- **reaction** (*int*) – MT reaction, e.g., 102 –> (n,gamma)
- **flag** (*Special*) – Isomeric state or fission yield distribution number Default is zero

Returns

- **xsVal** (*numpy.ndarray*) – Group-wise cross-section values
- **xsUnc** (*numpy.ndarray*) – Group-wise uncertainty values

:raises SerpentToolsException:: If the universe does not exist If the isotope's format is incorrect (not ZZAAA0/1)

5.2.8 serpentTools.ResultsReader

class serpentTools.**ResultsReader** (*filePath*)

Parser responsible for reading and working with result files.

When inspecting keys of *universes*, it is preferable to use an attribute based approach rather than positional. For example:

```
>>> for key in res.universes:
...     break
>>> key.universe
# rather than
>>> key[0]
```

metadata

Dictionary with serpent descriptive variables as keys and the corresponding description. Within the _res.m file this data is printed multiple times, but contains the same description and thus is stored only once. e.g., 'version': 'Serpent 2.1.29'

Type dict

resdata

Dictionary with serpent time-dependent variables as keys and the corresponding values. The data is unique for each burnup step. Some variables also contain uncertainties. e.g., 'absKeff': [[9.91938E-01, 0.00145],[1.81729E-01, 0.00240]]

Type dict

universes

Dictionary of universe identifiers `UnivTuple` and their corresponding *HomogUniv* objects. The keys describe a unique state: 'universe', burnup (MWd/kg), burnup index, time (days) ('0', 0.0, 0, 0.0). Burnup indexes are zero-indexed, meaning the first step is index 0.

Type dict

Parameters **filePath** (*str*) – path to the results file

Raises

- **SerpentToolsException** – Serpent version is not supported, No universes are found in the file, No results are collected, Corrupted results
- **IOError** – file is unexpectedly closes while reading:

compareMetadata(*other*, *header=False*)
Return True if the metadata (settings) are identical.

Parameters

- **other** (*ResultsReader*) – Class against which to compare
- **header** (*bool*) – Print/log an `info` message about this comparison.

Returns If the metadata are identical

Return type bool

Raises **TypeError** – If `other` is not of the same class as this class nor a subclass of this class

compareResults(*other*, *lower=0*, *upper=10*, *sigma=2*, *header=False*)
Compare the contents of the results dictionary

Parameters

- **other** (*ResultsReader*) – Class against which to compare
- **lower** (*float or int*) – Lower limit for relative tolerances in percent Differences below this will be considered allowable
- **upper** (*float or int*) – Upper limit for relative tolerances in percent. Differences above this will be considered failure and errors messages will be raised
- **sigma** (*int*) – Size of confidence interval to apply to quantities with uncertainties. Quantities that do not have overlapping confidence intervals will fail
- **header** (*bool*) – Print/log an `info` message about this comparison.

Returns If the results data agree to given tolerances

Return type bool

Raises **TypeError** – If `other` is not of the same class as this class nor a subclass of this class

compareUniverses(*other*, *lower=0*, *upper=10*, *sigma=2*)
Compare the contents of the `universes` dictionary

Parameters

- **other** (*ResultsReader*) – Reader by which to compare
- **lower** (*float or int*) – Lower limit for relative tolerances in percent Differences below this will be considered allowable
- **upper** (*float or int*) – Upper limit for relative tolerances in percent. Differences above this will be considered failure and errors messages will be raised
- **sigma** (*int*) – Size of confidence interval to apply to quantities with uncertainties. Quantities that do not have overlapping confidence intervals will fail

Returns If the contents of the universes agree to given tolerances

Return type bool

Raises **TypeError** – If `other` is not of the same class as this class nor a subclass of this class

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

Parameters

- **univ** (*str*) – Unique str for the desired universe
- **burnup** (*float or int, optional*) – Burnup [MWd/kgU] of the desired universe
- **timeDays** (*float or int, optional*) – Time [days] of the desired universe
- **index** (*int, optinal*) – Point of interest in the burnup/days index

Returns Requested universe

Return type *HomogUniv*

:raises KeyError:: If the requested universe could not be found :raises ~serpentTools.SerpentToolsException: If burnup, days and index are not given

plot(*x*, *y=None*, *right=None*, *sigma=3*, *ax=None*, *legend=None*, *ncol=None*, *xlabel=True*, *ylabel=None*, *logx=False*, *logy=False*, *loglog=False*, *rightlabel=None*)
Plot quantities over time

Parameters

- **x** (*str or iterable of strings*) – `y` is not given, then plot these quantites against burnup in days. Otherwise, plot this quantity as the x axis with same rules as if called by `plot('burndays', x)`. Burnup options are `{'burnup', 'days', 'burnDays', 'burnStep'}`
- **y** (*str or iterable of strings*) – Quantity or quantities to plot. For all entries, only the first column, with respect to time, will be plotted. If the second column exists, and sigma is > 0, that column will be treated as the relative uncertainty for an errorbar plot. If a dictionary is passed, then plots will be labeled by the values of that dictionary, e.g. `{'anaKeff': $k_{eff}$}` would plot the first column of `anaKeff` with a LaTeX-ready k_{eff}
- **right** (*str or iterable of strings*) – Quantites to plot on the same plot, but with a different y axis and common x axis. Same rules apply as for arguments to `y`. Each label will modified to have a unique identifier indicating the plot uses the right y-axis
- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **sigma** (*int*) – Confidence interval to apply to errors. If not given or `0`, no errors will be drawn.
- **legend** (*bool or str or None*) – Automatically label the plot. No legend will be made if a single item is plotted. Pass one of the following values to place the legend outside the plot: above, right
- **ncol** (*int*) – Integer number of columns to apply to the legend.
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.

- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool or list or tuple*) – Apply a log scale to the y-axis. If passing values to `right`, this can be a two item list or tuple, corresponding to log-scaling the left and right axis, respectively.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **rlabel** (*str or None*) – If given and passing values to `right`, use this to label the y-axis.

Returns If right is not given, then only the primary axes object is returned. Otherwise, the primary and the "right" axes object are returned

Return type `matplotlib.axes.Axes` or tuple of axes

toMatlab(*fileP*, *reconvert=True*, *append=True*, *format='5'*, *longNames=True*, *compress=True*, *oned='row'*)

Write a binary MATLAB file from the contents of this reader

The group constant data will be presented as a multi-dimensional array, rather than a stacked 2D matrix. The axis are ordered `burnup, universeIndex, group, value/uncertainty`

The ordering of the universes can be found in the `'UNIVERSES'` vector if `reconvert==True`, otherwise `'universes'`. Each universe ID is present in this vector, ordered to their position along the second axis in the matrix.

Parameters

- **fileP** (*str or file-like object*) – Name of the file to write
- **reconvert** (*bool*) – If this evaluates to true, convert values back into their original form as they appear in the output file.
- **append** (*bool*) – If true and a file exists under `output`, append to that file. Otherwise the file will be overwritten
- **format** (*{'5', '4'}*) – Format of file to write. `'5'` for MATLAB 5 to 7.2, `'4'` for MATLAB 4
- **longNames** (*bool*) – If true, allow variable names to reach 63 characters, which works with MATLAB 7.6+. Otherwise, maximum length is 31 characters
- **compress** (*bool*) – If true, compress matrices on write
- **oned** (*{'row', 'col'}:*) – Write one-dimensional arrays as row vectors if `oned=='row'` (default), or column vectors

Examples

```
>>> import serpentTools
>>> from scipy.io import loadmat
>>> r = serpentTools.readDataFile('pwr_res.m')
# convert to original variable names
>>> r.toMatlab('pwr_res.mat', True)
>>> loaded = loadmat('pwr_res.mat')
>>> loaded['ABS_KEFF']
array([[0.991938, 0.00145 ],
       [0.181729, 0.0024  ]])
>>> kinf = loaded['INF_KINF']
>>> kinf.shape
```

(continues on next page)

(continued from previous page)

```
(2, 1, 1, 2)
>>> kinf[:, 0, 0, 0]
array([0.993385, 0.181451])
>>> tot = loaded['INF_TOT']
>>> tot.shape
(2, 1, 2, 2)
>>> tot[:, 0, :, 0]
array([[0.496553, 1.21388 ],
       [0.481875, 1.30993 ]])
# use the universes key to identify ordering of universes
>>> loaded['UNIVERSES']
array([0])
```

:raises ImportError:: If *scipy* is not installed

See also:

- `scipy.io.savemat()`

5.2.9 serpentTools.SensitivityReader

class serpentTools.**SensitivityReader**(*filePath*)

Class for reading sensitivity files

The arrays that are stored in *sensitivities* and *energyIntegratedSens* are stored under converted names. The original names from SERPENT are of the form `ADJ_PERT_KEFF_SENS` or `ADJ_PERT_KEFF_SENS_E_INT`, respectively. Since this reader stores the resulting arrays in unique locations, the names are converted to a succint form. The two arrays listed above would be stored both as `keff` in *sensitivities* and *energyIntegratedSens*. All names are converted to `mixedCaseNames` to fit the style of the project.

Ordered dictionaries *materials*, *zais* and *perts* contain keys of the names of their respective data, and the corresponding index, `iSENS_ZAI_zzaaai`, in the sensitivity arrays. These arrays are zero-indexed, so the first item will have an index of zero. The data stored in the *sensitivities* and *energyIntegratedSens* dictionaries has the exact same structure as if the arrays were loaded into `MATLAB`/`Octave`, but with zero-indexing.

The matrices in *sensitivities* are ordered as they would be in MATLAB. The five dimensions correspond to:

1. *materials* that were contained perturbed isotopes
2. *zais* that were perturbed
3. *perts* - reactions that were perturbed, e.g. `'total xs'`
4. *energies* - which energy group contained the perturbation. Will have one fewer dimensions than the number of values in *energies*, corresponding to the number of energy groups.
5. [value, relative uncertainty] pairs

The matrices in *energyIntegratedSens* will have the same structure, but with the *energies* dimension removed.

Note: Arrays generated using the history option `sens opt history 1` are not currently stored on the reader. See feature request #367

Parameters **filePath** (*str*) – Path to sensitivity file

nMat
Number of materials

Type None or int

nZai
Number of perturbed isotopes

Type None or int

nPert
Number of perturbations

Type None or int

nEne
Number of energy groups

Type None or int

nMu
Number of perturbed materials

Type None or int

materials
Ordered dictionary of materials that have been perturbed.

Type OrderedDict

zais
Ordered dictionary of nuclides that have been perturbed

Type OrderedDict

perts
Ordered dictionary of reactions that have been perturbed, e.g *'total xs'*

Type OrderedDict

latGen
Number of latent generations used to generate these sensitivities

Type int

energies
Array of energy bounds for the sensitivities, from lowest to highest

Type None or numpy.array

lethargyWidths
Array of lethargy widths of each energy group.

Type None or numpy.array

sensitivities
Dictionary of names of sensitivities and their corresponding arrays.

Type dict

energyIntegratedSens
Dictionary of names of the sensitivities that have been integrated against energy, and their corresponding arrays

Type dict

plot(*resp*, *zai=None*, *pert=None*, *mat=None*, *sigma=3*, *normalize=True*, *ax=None*, *labelFmt=None*, *title=None*, *logx=True*, *logy=False*, *loglog=False*, *xlabel=None*, *ylabel=None*, *legend=None*, *ncol=1*)

Plot sensitivities due to some or all perturbations.

Note: Without passing `zai`, `pert`, or `mat` arguments, this method will plot all permutations of sensitivities for a given response.

Parameters

- **resp** (*str*) – Name of the specific response to be examined. Must be a key in `sensitivities` and `energyIntegratedSens`
- **zai** (*None or str or int or iterable*) – Plot sensitivities due to these isotopes. Passing `None` will plot against all isotopes.
- **pert** (*None or str or list of strings*) – Plot sensitivities due to these perturbations. Passing `None` will plot against all perturbations.
- **mat** (*None or str or list of strings*) – Plot sensitivities due to these materials. Passing `None` will plot against all materials.
- **sigma** (*int*) – Confidence interval to apply to errors. If not given or `0`, no errors will be drawn.
- **normalize** (*True*) – Normalize plotted data per unit lethargy
- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **labelFmt** (*None or str*) – Formattable string to be applied to the labels. The following entries will be formatted for each plot permuation:

```
{m} - name of the material
{z} - isotope zai
{p} - specific perturbation
{r} - response being plotted
```

- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **legend** (*bool or str or None*) – Automatically label the plot. No legend will be made if a single item is plotted. Pass one of the following values to place the legend outside the plot: above, right
- **ncol** (*int*) – Integer number of columns to apply to the legend.

Returns Ax on which the data was plotted.

Return type matplotlib.axes.Axes

Raises **KeyError** – If response or any passed perturbation settings are not present on the object

See also:

- str.format() - used for formatting labels

toMatlab(*fileP*, *reconvert=True*, *append=True*, *format='5'*, *longNames=True*, *compress=True*, *oned='row'*)

Write a binary MATLAB file from the contents of this object

Parameters

- **fileP** (*str or file-like object*) – Name of the file to write. .mat extension is not needed if append==True
- **reconvert** (*bool*) – If this evaluates to true, convert values back into their original form as they appear in the output file.
- **append** (*bool*) – If true and a file exists under output, append to that file. Otherwise the file will be overwritten
- **format** (*{'5', '4'}*) – Format of file to write. '5' for MATLAB 5 to 7.2, '4' for MATLAB 4
- **longNames** (*bool*) – If true, allow variable names to reach 63 characters, which works with MATLAB 7.6+. Otherwise, maximum length is 31 characters
- **compress** (*bool*) – If true, compress matrices on write
- **oned** (*{'row', 'col'}*) – Write one-dimensional arrays as row vectors if oned=='row' (default), or column vectors

Raises **ImportError** – If *scipy* is not installed

See also:

- scipy.io.savemat()

5.2.10 serpentTools.XSPlotReader

class serpentTools.**XSPlotReader**(*filePath*)

Parser responsible for reading and working with xsplot output files. These files can be generated using:

http://serpent.vtt.fi/mediawiki/index.php/It_syntax_manual#set_xsplot

Parameters **filePath** (*str*) – path to the xsplot result file

xsections

Contains XSData objects with keys given by their names. There should be one XSData instance for each isotope and material present in the problem.

Type NamedDict

metadata

Contains data pertinent to all XSData instances collected. One particularly important entry is the 'egrid', which is a numpy array of values that define the bin structure XS were recorded in by Serpent. In addition, Serpent defines the 'majorant_xs', which is the L-inf norm among all macroscopic cross sections used in the problem. This gets used in the delta tracking routines usually.

Type dict

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

Type dict

CHAPTER

SIX

CONTAINERS

Many of the readers provided in this project and by *read()* rely on custom classes for storing physically-similar data. This includes detector data, where tallies can be stored next to underlying spatial and/or energy grids, or depleted material data with isotopes and burnup data. Below are links to reference documentation for all of the helper classes used in `serpentTools`

Note: With the exception of the detectors, these classes are not well suited to be created and populated outside of the readers. Much of the data accessing methods are stable, but providing data may change in future versions

6.1 Detectors

Warning: Serpent 1 detectors are not supported as of 0.8.0

Note: Energy grids are stored in order of increasing energy, exactly as they appear in the output file. The three columns of this array correspond to lower bound, upper bound, and mid-point of that energy group

Detector	Class for storing detector data with multiple bins
CartesianDetector	Class for storing detector data with Cartesian grid
HexagonalDetector	Class for storing detector data with a hexagonal grid
CylindricalDetector	Class for storing detector data with a cylindrical mesh
SphericalDetector	Class for storing detector data with multiple bins

6.1.1 serpentTools.Detector

class serpentTools.**Detector**(*name*, *bins=None*, *tallies=None*, *errors=None*, *indexes=None*, *grids=None*)

Class for storing detector data with multiple bins

For detectors with spatial meshes, including rectilinear, hexagonal, cylindrical, or spherical meshes, refer to companion classes *serpentTools.CartesianDetector*, *serpentTools.HexagonalDetector*, *serpentTools.CylindricalDetector*, or *serpentTools.SphericalDetector*

If simply the tally bins are available, it is recommended to use the *fromTallyBins()* class method. This will reshape the data and separate the mean tally [second to last column] and relative errors [last column].

Parameters

- **name** (*str*) – Name of this detector
- **bins** (*numpy.ndarray, optional*) – Full 2D tally data from detector file, including tallies and errors in last two columns
- **tallies** (*numpy.ndarray, optional*) – Reshaped tally data such that each dimension corresponds to a unique bin, such as energy or spatial bin.
- **errors** (*numpy.ndarray, optional*) – Reshaped error data such that each dimension corresponds to a unique bin, such as energy or spatial bin. Note: this is a relative error as it would appear in the output file
- **indexes** (*iterable of string, optional*) – Iterable naming the bins that correspond to reshaped `tally` and *errors*.
- **grids** (*dict, optional*) – Supplemental grids that may be supplied to this detector, including energy points or spatial coordinates.

name
Name of this detector

Type str

bins
Full 2D tally data from detector file, including tallies and errors in last two columns

Type numpy.ndarray or None

tallies
Reshaped tally data such that each dimension corresponds to a unique bin, such as energy or spatial bin.

Type numpy.ndarray or float or None

errors
Reshaped error data such that each dimension corresponds to a unique bin, such as energy or spatial bin. Note: this is a relative error as it would appear in the output file

Type numpy.ndarray or float or None

indexes
Iterable naming the bins that correspond to reshaped `tally` and *errors*. The tuple `(energy, ymesh, xmesh)` indicates that *tallies* should have three dimensions corresponding to various energy, y-position, and x-position bins. Must be set after *tallies* or *errors* and agree with the shape of each

Type tuple or None

grids
Dictionary containing grid information for binned quantities like energy or time.

Type dict

energy
Potential underlying energy grid in MeV. Will be `(n_ene, 3)`, where `n_ene` is the number of values in the energy grid. Each row `energy[j]` will be the low point, high point, and mid point of the energy bin `j`.

Type numpy.ndarray or None

Raises

- **ValueError** – If some spatial grid is found in `indexes` during creation. This class is ill-suited for these problems. Refer to the companion classes mentioned above.
- **IndexError** – If the shapes of `bins`, `tallies`, and `errors` are inconsistent

classmethod fromTallyBins(*name*, *bins*, *grids=None*)
Create a detector instance from 2D detector data

Parameters

- **name** (*str*) – Name of this detector
- **bins** (*numpy.ndarray*) – 2D array taken from Serpent. Expected to have either 12 or 13 columns, where the latter indicates a time bin has been added.
- **grids** (*dict, optional*) – Dictionary of underlying energy, space, and/or time data.

Returns

Return type *Detector*

Raises **ValueError** – If the tally data does not appear to be Serpent 2 tally data

meshPlot(*xdim*, *ydim*, *what='tallies'*, *fixed=None*, *ax=None*, *cmap=None*, *cbarLabel=None*, *logColor=False*, *xlabel=None*, *ylabel=None*, *logx=False*, *logy=False*, *loglog=False*, *title=None*, *thresh=None*, ***kwargs*)
Plot tally data as a function of two bin types on a cartesian mesh.

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'}*) – Color meshes from tally data or uncertainties
- **fixed** (*None or dict*) – Dictionary controlling the reduction in data down to one dimension
- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **cmap** (*str, optional*) – Valid Matplotlib colormap to apply to the plot.
- **logColor** (*bool*) – If true, apply a logarithmic coloring to the data positive data
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **title** (*str*) – Title to apply to the figure.
- **thresh** (*float*) – Do not plot data less than or equal to this value.
- **cbarLabel** (*str*) – Label to apply to colorbar. If not given, infer from `what`

- **kwargs** (*dict, optional*) – Addition keyword arguments to pass to `pcolormesh()`

Returns Ax on which the data was plotted.

Return type `matplotlib.axes.Axes`

Raises

- **serpentTools.SerpentToolsException** – If data to be plotted, with or without constraints, is not 1D
- **KeyError** – If `fixed` is given and `xdim` or `ydim` are contained in `fixed`
- **AttributeError** – If the data set by `what` not in the allowed selection
- **ValueError** – If the data contains negative quantities and `logColor` is `True`

See also:

- *slice()*
- `matplotlib.pyplot.pcolormesh()`

plot(*xdim=None*, *what='tallies'*, *sigma=None*, *fixed=None*, *ax=None*, *xlabel=None*, *ylabel=None*, *steps=False*, *labels=None*, *logx=False*, *logy=False*, *loglog=False*, *legend=None*, *ncol=1*, *title=None*, ***kwargs*)

Simple plot routine for 1- or 2-D data

Parameters

- **xdim** (*str, optional*) – Plot the data corresponding to changing this bin, e.g. `"energy"`. Must exist in *indexes*
- **what** (*{'tallies', 'errors'}*) – Primary data to plot
- **sigma** (*int*) – Confidence interval to apply to errors. If not given or `0`, no errors will be drawn.
- **fixed** (*None or dict*) – Dictionary controlling the reduction in data down to one dimension
- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label. If `xdim` is given and `xlabel` is `None`, then `xdim` will be applied to the x-axis.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **steps** (*bool*) – If true, plot the data as constant inside the respective bins. Sets `drawstyle` to be `steps-post` unless `drawstyle` given in `kwargs`
- **labels** (*None or iterable*) – Labels to apply to each line drawn. This can be used to identify which bin is plotted as what line.
- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.

- **legend** (*bool or str or None*) – Automatically label the plot. No legend will be made if a single item is plotted. Pass one of the following values to place the legend outside the plot: above, right
- **ncol** (*int*) – Integer number of columns to apply to the legend.
- **title** (*str*) – Title to apply to the figure.
- **kwargs** (*dict, optional*) – Addition keyword arguments to pass to `plot()` or `errorbar()` function.

Returns Ax on which the data was plotted.

Return type `matplotlib.axes.Axes`

Raises

- **SerpentToolsException** – If data contains more than 2 dimensions
- **AttributeError** – If plot data or *indexes* set up.

See also:

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

reshapedBins()

Obtain multi-dimensional tally, error, and index data

Returns

- **tallies** (*numpy.ndarray*) – Potentially multi-dimensional array corresponding to tally data along each bin index
- **errors** (*numpy.ndarray*) – Potentially multi-dimensional array corresponding to tally relative error along each bin index
- **indexes** (*list of str*) – Ordering of named bin information, e.g. `"xmesh"`, `"energy"`, corresponding to axis in `tallies` and `errors`

Examples

A detector is created with a single bin with two bins values. These could represent tallying two different reaction rates

```
>>> import numpy
>>> from serpentTools import Detector
>>> bins = numpy.ones((2, 12))
>>> bins[1, 0] = 2
>>> bins[1, 4] = 2
>>> bins[:, -2:] = [
...     [5.0,  0.1],
...     [10.0, 0.2]]
>>> det = Detector("reshape", bins=bins)
>>> tallies, errors, indexes = det.reshapedBins()
>>> tallies
array([5.0, 10.0])
>>> errors
array([0.1, 0.2])
```

(continues on next page)

(continued from previous page)

```
>>> indexes
["reaction", ]
```

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

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

Parameters

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

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

Return type numpy.ndarray

Raises **AttributeError** – If `data` is not supported

spectrumPlot (*fixed=None*, *ax=None*, *normalize=True*, *xlabel=None*, *ylabel=None*, *steps=True*, *logx=True*, *logy=False*, *loglog=False*, *sigma=3*, *labels=None*, *legend=None*, *ncol=1*, *title=None*, ***kwargs*)

Quick plot of the detector value as a function of energy.

Parameters

- **fixed** (*None or dict*) – Dictionary controlling the reduction in data
- **ax** (matplotlib.axes.Axes, optional) – Ax on which to plot the data. If not provided, create a new plot
- **normalize** (*bool*) – Normalize quantities per unit lethargy
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **steps** (*bool*) – Plot tally as constant inside bin
- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **sigma** (*int*) – Confidence interval to apply to errors. If not given or `0`, no errors will be drawn.
- **labels** (*None or iterable*) – Labels to apply to each line drawn. This can be used to identify which bin is plotted as what line.
- **legend** (*bool or str or None*) – Automatically label the plot. No legend will be made if a single item is plotted. Pass one of the following values to place the legend outside the plot: above, right
- **ncol** (*int*) – Integer number of columns to apply to the legend.
- **title** (*str*) – Title to apply to the figure.

- **kwargs** (*dict, optional*) – Addition keyword arguments to pass to `matplotlib.pyplot.plot()` or `matplotlib.pyplot.errorbar()`

Returns Ax on which the data was plotted.

Return type `matplotlib.axes.Axes`

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

See also:

- *slice()*

toMatlab(*fileP*, *reconvert=True*, *append=True*, *format='5'*, *longNames=True*, *compress=True*, *oned='row'*)

Write a binary MATLAB file from the contents of this object

Parameters

- **fileP** (*str or file-like object*) – Name of the file to write. `.mat` extension is not needed if `append==True`
- **reconvert** (*bool*) – If this evaluates to true, convert values back into their original form as they appear in the output file.
- **append** (*bool*) – If true and a file exists under `output`, append to that file. Otherwise the file will be overwritten
- **format** (*{'5', '4'}*) – Format of file to write. `'5'` for MATLAB 5 to 7.2, `'4'` for MATLAB 4
- **longNames** (*bool*) – If true, allow variable names to reach 63 characters, which works with MATLAB 7.6+. Otherwise, maximum length is 31 characters
- **compress** (*bool*) – If true, compress matrices on write
- **oned** (*{'row', 'col'}*) – Write one-dimensional arrays as row vectors if `oned=='row'` (default), or column vectors

Raises **ImportError** – If *scipy* is not installed

See also:

- `scipy.io.savemat()`

6.1.2 serpentTools.CartesianDetector

class serpentTools.**CartesianDetector**(*name*, *bins=None*, *tallies=None*, *errors=None*, *indexes=None*, *grids=None*, *x=None*, *y=None*, *z=None*)

Class for storing detector data with Cartesian grid

If simply the tally bins are available, it is recommended to use the `fromTallyBins()` class method. This will reshape the data and separate the mean tally [second to last column] and relative errors [last column].

Note: In order to get full functionality from this class, *x*, *y*, and *z* must be set. All grids are expected to be `(N, 3)` arrays, not necessarily of equal shape.

Parameters

- **name** (*str*) – Name of this detector
- **bins** (*numpy.ndarray*) – Full 2D tally data from detector file, including tallies and errors in last two columns
- **tallies** (*numpy.ndarray*) – Reshaped tally data such that each dimension corresponds to a unique bin, such as energy or spatial bin.
- **errors** (*numpy.ndarray*) – Reshaped error data such that each dimension corresponds to a unique bin, such as energy or spatial bin. Note: this is a relative error as it would appear in the output file
- **indexes** (*dict or None*) – Dictionary mapping the bin name to its corresponding axis in *tallies* and *errors*, e.g. `{"energy": 0}`. Optional
- **x** (*numpy.ndarray, optional*) – Directly set the x grid
- **y** (*numpy.ndarray, optional*) – Directly set the y grid
- **z** (*numpy.ndarray, optional*) – Directly set the z grid
- **grids** (*dict, optional*) – Supplemental grids that may be supplied to this detector, including energy points or spatial coordinates. If spatial grids are not provided directly with the `x`, `y`, or `z` arguments, the grids will be pulled from this dictionary in the following manner.
  - Key `"X"` denotes the *x* grid
  - Key `"Y"` denotes the *y* grid
  - Key `"Z"` denotes the *z* grid

  If the keys are not present, or the grids are directly provided, no actions are taken.

name
Name of this detector

Type str

bins
Full 2D tally data from detector file, including tallies and errors in last two columns

Type numpy.ndarray or None

tallies
Reshaped tally data such that each dimension corresponds to a unique bin, such as energy or spatial bin.

Type numpy.ndarray or None

errors
Reshaped error data such that each dimension corresponds to a unique bin, such as energy or spatial bin. Note: this is a relative error as it would appear in the output file

Type numpy.ndarray or None

indexes
Dictionary mapping the bin name to its corresponding axis in *tallies* and *errors*, e.g. `{"energy": 0}`.

Type dict

energy
Potential underlying energy grid in MeV. Will be `(n_ene, 3)`, where `n_ene` is the number of values in the energy grid. Each row `energy[j]` will be the low point, high point, and mid point of the energy bin `j`.

Type numpy.ndarray or None

x

X grid

Type numpy.ndarray or None

y

Y grid

Type numpy.ndarray or None

z

Z grid

Type numpy.ndarray or None

Raises **ValueError** – If the values for `x`, `y`, and `z` are not 2D arrays with 3 columns

meshPlot(*xdim='x'*, *ydim='y'*, *what='tallies'*, *fixed=None*, *ax=None*, *cmap=None*, *cbarLabel=None*, *logColor=False*, *xlabel=None*, *ylabel=None*, *logx=False*, *logy=False*, *loglog=False*, *title=None*, *thresh=None*, ***kwargs*)

Plot tally data as a function of two bin types on a cartesian mesh.

Parameters

- **xdim** (*str*) – Primary dimension - will correspond to x-axis on plot. Defaults to "x"
- **ydim** (*str*) – Secondary dimension - will correspond to y-axis on plot. Defaults to "y"
- **what** (*{'tallies', 'errors'}*) – Color meshes from tally data or uncertainties
- **fixed** (*None or dict*) – Dictionary controlling the reduction in data down to one dimension
- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **cmap** (*str, optional*) – Valid Matplotlib colormap to apply to the plot.
- **logColor** (*bool*) – If true, apply a logarithmic coloring to the data positive data
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **title** (*str*) – Title to apply to the figure.
- **thresh** (*float*) – Do not plot data less than or equal to this value.
- **cbarLabel** (*str*) – Label to apply to colorbar. If not given, infer from `what`
- **kwargs** (*dict, optional*) – Addition keyword arguments to pass to `pcolormesh()`

Returns Ax on which the data was plotted.

Return type matplotlib.axes.Axes

Raises

- **serpentTools.SerpentToolsException** – If data to be plotted, with or without constraints, is not 1D
- **KeyError** – If `fixed` is given and `xdim` or `ydim` are contained in `fixed`
- **AttributeError** – If the data set by `what` not in the allowed selection
- **ValueError** – If the data contains negative quantities and `logColor` is `True`

See also:

- `slice()`
- `matplotlib.pyplot.pcolormesh()`

6.1.3 serpentTools.HexagonalDetector

class serpentTools.**HexagonalDetector**(*name*, *bins=None*, *tallies=None*, *errors=None*, *indexes=None*, *z=None*, *centers=None*, *pitch=None*, *hexType=None*, *grids=None*)

Class for storing detector data with a hexagonal grid

If simply the tally bins are available, it is recommended to use the `fromTallyBins()` class method. This will reshape the data and separate the mean tally [second to last column] and relative errors [last column].

Note: In order to get full functionality from this class, *z*, `centers`, *pitch*, and *hexType* must be set. This can be done by having `"Z"` and `"COORDS"` as keys in `grids` and passing `pitch` and `hexType` directly, or by setting the attributes directly. Z grid is expected to be `(N, 3)` array.

Parameters

- **name** (*str*) – Name of this detector
- **bins** (*numpy.ndarray, optional*) – Full 2D tally data from detector file, including tallies and errors in last two columns
- **tallies** (*numpy.ndarray, optional*) – Reshaped tally data such that each dimension corresponds to a unique bin, such as energy or spatial bin.
- **errors** (*numpy.ndarray, optional*) – Reshaped error data such that each dimension corresponds to a unique bin, such as energy or spatial bin. Note: this is a relative error as it would appear in the output file
- **indexes** (*dict, optional*) – Dictionary mapping the bin name to its corresponding axis in *tallies* and *errors*, e.g. `{"energy": 0}`
- **centers** (*numpy.ndarray, optional*) – `(N, 2)` array with centers of each hexagonal element
- **pitch** (*float, optional*) – Center to center distance between adjacent hexagons
- **hexType** (*{2, 3}, optional*) – Integer orientation, identical to Serpent detector structure. 2 corresponds to a hexagon with flat faces perpendicular to y-axis. 3 corresponds to a flat faces perpendicular to x-axis

- **grids** (*dict, optional*) – Supplemental grids that may be supplied to this detector, including energy points or spatial coordinates.

name
Name of this detector

Type str

bins
Full 2D tally data from detector file, including tallies and errors in last two columns

Type numpy.ndarray or None

tallies
Reshaped tally data such that each dimension corresponds to a unique bin, such as energy or spatial bin.

Type numpy.ndarray or None

pitch
Center to center distance between hexagons. Must be set before calling *hexPlot()*

Type float or None

hexType
Integer orientation, identical to Serpent detector structure. 2 corresponds to a hexagon with flat faces perpendicular to y-axis. 3 corresponds to a flat faces perpendicular to x-axis

Type {2, 3} or None

errors
Reshaped error data such that each dimension corresponds to a unique bin, such as energy or spatial bin. Note: this is a relative error as it would appear in the output file

Type numpy.ndarray or None

indexes
Dictionary mapping the bin name to its corresponding axis in *tallies* and *errors*, e.g. `{"energy": 0}`.

Type dict or None

energy
Potential underlying energy grid in MeV. Will be `(n_ene, 3)`, where `n_ene` is the number of values in the energy grid. Each row `energy[j]` will be the low point, high point, and mid point of the energy bin `j`.

Type numpy.ndarray or None

coords
Centers of hexagonal meshes in XY plane

Type numpy.ndarray or None

z
Z Grid

Type numpy.ndarray or None

hexPlot(*what='tallies'*, *fixed=None*, *ax=None*, *cmap=None*, *thresh=None*, *logColor=False*, *xlabel=None*, *ylabel=None*, *logx=False*, *logy=False*, *loglog=False*, *title=None*, *normalizer=None*, *cbarLabel=None*, *borderpad=2.5*, ***kwargs*)
Create and return a hexagonal mesh plot.

Parameters

- **what** (*{'tallies', 'errors'}, optional*) – Quantity to plot. Defaults to "tallies"
- **fixed** (*dict, optional.*) – Dictionary of slicing arguments to pass to slice(). If not provided, detector must be defined with a 2D grid.
- **ax** (matplotlib.axes.Axes, optional) – Ax on which to plot the data. If not provided, create a new plot
- **thresh** (*float, optional*) – Threshold value for plotting values. A hexagon must have a value greater than this quantity in order to be drawn.
- **cmap** (*str, optional*) – Valid Matplotlib colormap to apply to the plot.
- **{logColor}** –
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as None, a label will be determined from other arguments. If not None but evaluates to False, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as None, a label will be determined from other arguments. If not None but evaluates to False, do not label.
- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **title** (*str*) – Title to apply to the figure.
- **borderpad** (*int or float, optional*) – Percentage of total plot to apply as a border. A value of zero means that the extreme edges of the hexagons will touch the x and y axis.
- **kwargs** (*dict, optional*) – Addition keyword arguments to pass to matplotlib.patches.RegularPolygon

Raises **AttributeError** – If *pitch* and *hexType* are not set.

meshPlot(*xdim*, *ydim*, *what='tallies'*, *fixed=None*, *ax=None*, *cmap=None*, *logColor=False*, *xlabel=None*, *ylabel=None*, *logx=False*, *logy=False*, *loglog=False*, *title=None*, ***kwargs*)

Plot tally data as a function of two bin types on a cartesian mesh.

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'}*) – Color meshes from tally data or uncertainties
- **fixed** (*None or dict*) – Dictionary controlling the reduction in data down to one dimension
- **ax** (matplotlib.axes.Axes, optional) – Ax on which to plot the data. If not provided, create a new plot
- **cmap** (*str, optional*) – Valid Matplotlib colormap to apply to the plot.
- **logColor** (*bool*) – If true, apply a logarithmic coloring to the data positive data
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as None, a label will be determined from other arguments. If not None but evaluates to False, do not label.

- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **title** (*str*) – Title to apply to the figure.
- **thresh** (*float*) – Do not plot data less than or equal to this value.
- **cbarLabel** (*str*) – Label to apply to colorbar. If not given, infer from `what`
- **kwargs** (*dict, optional*) – Addition keyword arguments to pass to `pcolormesh()`

Returns Ax on which the data was plotted.

Return type `matplotlib.axes.Axes`

Raises

- **serpentTools.SerpentToolsException** – If data to be plotted, with or without constraints, is not 1D
- **KeyError** – If `fixed` is given and `xdim` or `ydim` are contained in `fixed`
- **AttributeError** – If the data set by `what` not in the allowed selection
- **ValueError** – If the data contains negative quantities and `logColor` is `True`

See also:

- `slice()`
- `matplotlib.pyplot.pcolormesh()`

6.1.4 serpentTools.CylindricalDetector

class serpentTools.**CylindricalDetector**(*name*, *bins=None*, *tallies=None*, *errors=None*, *indexes=None*, *grids=None*)

Class for storing detector data with a cylindrical mesh

If simply the tally bins are available, it is recommended to use the `fromTallyBins()` class method. This will reshape the data and separate the mean tally [second to last column] and relative errors [last column].

Parameters

- **name** (*str*) – Name of this detector
- **bins** (*numpy.ndarray, optional*) – Full 2D tally data from detector file, including tallies and errors in last two columns
- **tallies** (*numpy.ndarray, optional*) – Reshaped tally data such that each dimension corresponds to a unique bin, such as energy or spatial bin.
- **errors** (*numpy.ndarray, optional*) – Reshaped error data such that each dimension corresponds to a unique bin, such as energy or spatial bin. Note: this is a relative error as it would appear in the output file
- **indexes** (*collections.OrderedDict, optinal*) – Dictionary mapping the bin name to its corresponding axis in *tallies* and *errors*, e.g. `{"energy": 0}`

- **grids** (*dict, optional*) – Supplemental grids that may be supplied to this detector, including energy points or spatial coordinates.

name
Name of this detector

Type str

bins
Full 2D tally data from detector file, including tallies and errors in last two columns

Type numpy.ndarray

tallies
Reshaped tally data such that each dimension corresponds to a unique bin, such as energy or spatial bin.

Type numpy.ndarray

errors
Reshaped error data such that each dimension corresponds to a unique bin, such as energy or spatial bin. Note: this is a relative error as it would appear in the output file

Type numpy.ndarray

indexes
Dictionary mapping the bin name to its corresponding axis in *tallies* and *errors*, e.g. `{"energy": 0}`.

Type collections.OrderedDict

energy
Potential underlying energy grid in MeV. Will be `(n_ene, 3)`, where `n_ene` is the number of values in the energy grid. Each row `energy[j]` will be the low point, high point, and mid point of the energy bin `j`.

Type numpy.ndarray or None

meshPlot(*xdim*, *ydim*, *what='tallies'*, *fixed=None*, *ax=None*, *cmap=None*, *logColor=False*, *xlabel=None*, *ylabel=None*, *logx=False*, *logy=False*, *loglog=False*, *title=None*, ***kwargs*)
Plot tally data as a function of two bin types on a cartesian mesh.

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'}*) – Color meshes from tally data or uncertainties
- **fixed** (*None or dict*) – Dictionary controlling the reduction in data down to one dimension
- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **cmap** (*str, optional*) – Valid Matplotlib colormap to apply to the plot.
- **logColor** (*bool*) – If true, apply a logarithmic coloring to the data positive data
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.

- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **title** (*str*) – Title to apply to the figure.
- **thresh** (*float*) – Do not plot data less than or equal to this value.
- **cbarLabel** (*str*) – Label to apply to colorbar. If not given, infer from `what`
- **kwargs** (*dict, optional*) – Addition keyword arguments to pass to `pcolormesh()`

Returns Ax on which the data was plotted.

Return type `matplotlib.axes.Axes`

Raises

- **serpentTools.SerpentToolsException** – If data to be plotted, with or without constraints, is not 1D
- **KeyError** – If `fixed` is given and `xdim` or `ydim` are contained in `fixed`
- **AttributeError** – If the data set by `what` not in the allowed selection
- **ValueError** – If the data contains negative quantities and `logColor` is `True`

See also:

- `slice()`
- `matplotlib.pyplot.pcolormesh()`

6.1.5 serpentTools.SphericalDetector

class serpentTools.**SphericalDetector**(*name*, *bins=None*, *tallies=None*, *errors=None*, *indexes=None*, *grids=None*)

Class for storing detector data with multiple bins

If simply the tally bins are available, it is recommended to use the `fromTallyBins()` class method. This will reshape the data and separate the mean tally [second to last column] and relative errors [last column].

Parameters

- **name** (*str*) – Name of this detector
- **bins** (*numpy.ndarray*) – Full 2D tally data from detector file, including tallies and errors in last two columns
- **tallies** (*numpy.ndarray*) – Reshaped tally data such that each dimension corresponds to a unique bin, such as energy or spatial bin.
- **errors** (*numpy.ndarray*) – Reshaped error data such that each dimension corresponds to a unique bin, such as energy or spatial bin. Note: this is a relative error as it would appear in the output file
- **indexes** (*dict*) – Dictionary mapping the bin name to its corresponding axis in *tallies* and *errors*, e.g. `{"energy": 0}`. Optional
- **grids** (*dict*) – Supplemental grids that may be supplied to this detector, including energy points or spatial coordinates.

name
Name of this detector

Type str

bins
Full 2D tally data from detector file, including tallies and errors in last two columns

Type numpy.ndarray

tallies
Reshaped tally data such that each dimension corresponds to a unique bin, such as energy or spatial bin.

Type numpy.ndarray

errors
Reshaped error data such that each dimension corresponds to a unique bin, such as energy or spatial bin. Note: this is a relative error as it would appear in the output file

Type numpy.ndarray

indexes
Dictionary mapping the bin name to its corresponding axis in *tallies* and *errors*, e.g. `{"energy": 0}`.

Type dict

energy
Potential underlying energy grid in MeV. Will be `(n_ene, 3)`, where `n_ene` is the number of values in the energy grid. Each row `energy[j]` will be the low point, high point, and mid point of the energy bin `j`.

Type numpy.ndarray or None

meshPlot(*xdim*, *ydim*, *what='tallies'*, *fixed=None*, *ax=None*, *cmap=None*, *logColor=False*, *xlabel=None*, *ylabel=None*, *logx=False*, *logy=False*, *loglog=False*, *title=None*, ***kwargs*)
Plot tally data as a function of two bin types on a cartesian mesh.

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'}*) – Color meshes from tally data or uncertainties
- **fixed** (*None or dict*) – Dictionary controlling the reduction in data down to one dimension
- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **cmap** (*str, optional*) – Valid Matplotlib colormap to apply to the plot.
- **logColor** (*bool*) – If true, apply a logarithmic coloring to the data positive data
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **logx** (*bool*) – Apply a log scale to x axis.

- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **title** (*str*) – Title to apply to the figure.
- **thresh** (*float*) – Do not plot data less than or equal to this value.
- **cbarLabel** (*str*) – Label to apply to colorbar. If not given, infer from `what`
- **kwargs** (*dict, optional*) – Addition keyword arguments to pass to `pcolormesh()`

Returns Ax on which the data was plotted.

Return type `matplotlib.axes.Axes`

Raises

- **serpentTools.SerpentToolsException** – If data to be plotted, with or without constraints, is not 1D
- **KeyError** – If `fixed` is given and `xdim` or `ydim` are contained in `fixed`
- **AttributeError** – If the data set by `what` not in the allowed selection
- **ValueError** – If the data contains negative quantities and `logColor` is `True`

See also:

- `slice()`
- `matplotlib.pyplot.pcolormesh()`

6.2 Supporting Classes

BranchContainer	Class that stores data for a single branch.
DepletedMaterial	Base class for storing material data from a depleted material file
HomogUniv	Class for storing homogenized universe specifications and retrieving them
XSData	Base class for storing cross section data an xsplot file

6.2.1 serpentTools.objects.BranchContainer

class serpentTools.objects.**BranchContainer**(*filePath*, *branchID*, *branchNames*, *stateData*)

Class that stores data for a single branch.

This container acts like a dictionary, mapping `UnivTuple` to *HomogUniv* corresponding to a specific branch state. As such, there is no need for the *universes* dictionary that previously served this purpose. However, it is provided for compatibility but will be removed after 0.9.0

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

- **filePath** (*str*) – Path to input file from which this container was connected
- **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
Name: value pairs for the variables defined on each branch card

Type dict

universes
Deprecated since version 0.8.0: Treat this object as the universes dictionary instead

Type dict

__setitem__(*key*, *value*)
Set self[key] to value.

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

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

Parameters

- **univID** (*str*) – Unique ID for the desired universe
- **burnup** (*float, optional*) – Burnup [MWd/kgU] of the desired universe
- **index** (*int, optional*) – Point of interest in the burnup index
- **days** (*float, optional*) – Point in time [d] for the desired universe

Returns Requested universe

Return type HomogUniv

Raises

- **KeyError** – If the requested universe could not be found
- **serpentTools.SerpentToolsException** – If neither burnup nor index are given

property orderedUniv
Universe keys sorted by ID and by burnup

update(*other*)
Update with contents of another BranchContainer

6.2.2 serpentTools.objects.DepletedMaterial

class serpentTools.objects.**DepletedMaterial**(*name*, *metadata*)
Base class for storing material data from a depleted material file

While `adens`, `mdens`, and `burnup` are accessible directly with `material.adens`, all variables read in from the file can be accessed through the `data` dictionary:

```
>>> assert material.adens is material.data['adens']
>>> assert material.adens is material['adens']
# The three methods are equivalent
```

Parameters

- **name** (*str*) – Name of this material
- **metadata** (*dict*) – Dictionary with file metadata

data
dictionary that stores all variable data

Type dict

zai
Isotopic ZZAAA identifiers, e.g. 93325

Changed in version 0.5.1: Now a list of integers, not strings

Type list

names
Names of isotopes, e.g. U235

Type list

days
Vector of total, cumulative days of burnup for the run that created this material

Type numpy.ndarray

burnup
Vector of total, cumulative burnup [MWd/kgU] for this specific material

Type numpy.ndarray

adens
2D array of atomic densities where where row `j` corresponds to isotope `j` and column `i` corresponds to time `i`

Type numpy.ndarray

mdens
2D array of mass densities where where row `j` corresponds to isotope `j` and column `i` corresponds to time `i`

Type numpy.ndarray

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

plot(*xUnits*, *yUnits=None*, *timePoints=None*, *names=None*, *zai=None*, *ax=None*, *legend=None*, *xlabel=None*, *ylabel=None*, *logx=False*, *logy=False*, *loglog=False*, *labelFmt=None*, *ncol=1*, *title=None*, ***kwargs*)
Plot some data as a function of time for some or all isotopes.

Note: `kwargs` will be passed to the plot for all isotopes. If `c='r'` is passed, to make a plot red, then data for all isotopes plotted will be red and potentially very confusing.

Parameters

- **xUnits** (*str*) – name of x value to obtain, e.g. `'days'`, `'burnup'` If `xUnits` is given and `yUnits` is `None`, then the plotted data will be `xUnits` against `'days'`
- **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

  Deprecated since version 0.7.0: Will plot against all time points
- **names** (*str or list or None*) – If given, plot values corresponding to these isotope names. Otherwise, plot values for all isotopes.
- **zai** (*int or list or None*) – If given, plot values corresponding to these isotope `ZZAAAI` values. Otherwise, plot for all isotopes

  New in version 0.5.1.
- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **legend** (*bool or str or None*) – Automatically label the plot. No legend will be made if a single item is plotted. Pass one of the following values to place the legend outside the plot: above, right
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label. Otherwise, use `xUnits`
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label. Otherwise, use `yUnits`
- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **labelFmt** (*str, optional*) – Formattable string for labeling the individual plots. If not given, just label as isotope name, e.g. `'U235'`. Will make the following substitutions on the `labelFmt` string, if given:

Keyword	Replacement
`'mat'`	name of this material
`'iso'`	specific isotope name
`'zai'`	specific isotope ZZAAAI

- **ncol** (*int*) – Integer number of columns to apply to the legend.
- **title** (*str*) – Title to apply to the figure.
- **kwargs** (*dict, optional*) – Addition keyword arguments to pass to `matplotlib.pyplot.plot()`

Returns Ax on which the data was plotted.

Return type `matplotlib.axes.Axes`

See also:

- `getValues()`

- `matplotlib.pyplot.plot()`
- `str.format()` - used for formatting labels

Raises

- **KeyError** – If x axis units are not `'days'` nor `'burnup'`
- **TypeError** – If both `names` and `zai` are given

6.2.3 serpentTools.objects.HomogUniv

class serpentTools.objects.**HomogUniv**(*name*, *bu*, *step*, *day*)

Class for storing homogenized universe specifications and retrieving them

Parameters

- **name** (*str*) – name of the universe
- **bu** (*float*) – burnup value
- **step** (*float*) – temporal step
- **day** (*float*) – depletion day

name

name of the universe

Type str

bu

non-negative burnup value

Type float or int

step

temporal step

Type int

day

non-negative depletion day

Type float or int

infExp

Expected values for infinite medium group constants

Type dict

infUnc

Relative uncertainties for infinite medium group constants

Type dict

b1Exp

Expected values for leakage corrected group constants

Type dict

b1Unc

Relative uncertainties for leakage-corrected group constants

Type dict

gc
Expected values for group constants that do not fit the `INF_` nor `B1_` pattern

Type dict

gcUnc
Relative uncertainties for group constants that do not fit the `INF_` nor `B1_` pattern

Type dict

reshaped
`True` if scattering matrices have been reshaped to square matrices. Otherwise, these matrices are stored as vectors.

Type bool

groups
Group boundaries from highest to lowest

Type None or `numpy.array`

numGroups
Number of energy groups bounded by `groups`

Type None or int

microGroups
Micro group structure used to produce group constants. Listed from lowest to highest

Type None or `numpy.array`

Raises **serpentTools.SerpentToolsException** – If a negative value of burnup, step, or burnup days is passed

__bool__()
Return True if data is stored on the object.

__getitem__(*gcname*)
Return just the group constant with this name, no uncertainty

To return data with uncertainties, or to return uncertainties, use *get()*.

__nonzero__()
Return True if data is stored on the object.

__setitem__(*gckey*, *gcvalue*)
Set the expected value of gckey to be gcvalue

No conversions are placed on the variable name. What you pass is what gets set.

For uncertainties, or to convert variable names to `mixedCase`, use *addData()*.

Much like a dictionary, this will overwrite existing data.

__str__()
Return str(self).

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

New in version 0.5.0: Reshapes scattering matrices according to setting `xs.reshapeScatter`. Matrices are of the form $S[i, j] = \Sigma_{s,i\rightarrow j}$

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

Parameters

- **variableName** (*str*) – Variable Name
- **variableValue** – Variable Value
- **uncertainty** (*bool*) – Set to `True` if this data is an uncertainty

Raises **TypeError** – If the uncertainty flag is not boolean

compareAttributes(*other*, *lower=0*, *upper=10*, *sigma=2*)
Compare attributes like group structure and burnup. Return the result

Parameters

- **other** (*HomogUniv*) – Universe against which to compare
- **lower** (*float or int*) – Lower limit for relative tolerances in percent Differences below this will be considered allowable
- **upper** (*float or int*) – Upper limit for relative tolerances in percent. Differences above this will be considered failure and errors messages will be raised
- **sigma** (*int*) – Size of confidence interval to apply to quantities with uncertainties. Quantities that do not have overlapping confidence intervals will fail

Returns `True` if the attributes agree within specifications

Return type bools

Raises **TypeError** – If `other` is not of the same class as this class nor a subclass of this class

compareB1Data(*other*, *sigma*)
Return `True` if contents of `b1Exp` and `b1Unc` agree

Parameters

- **other** (*HomogUniv*) – Object from which to grab group constant dictionaries
- **sigma** (*int*) – Size of confidence interval to apply to quantities with uncertainties. Quantities that do not have overlapping confidence intervals will fail
- **Returns** –
- **bool** – If the dictionaries contain identical values with uncertainties, and if those values have overlapping confidence intervals

Raises **TypeError** – If `other` is not of the same class as this class nor a subclass of this class

compareGCData(*other*, *sigma*)
Return `True` if contents of `gcExp` and `gcUnc` agree

Parameters

- **other** (*HomogUniv*) – Object from which to grab group constant dictionaries
- **sigma** (*int*) – Size of confidence interval to apply to quantities with uncertainties. Quantities that do not have overlapping confidence intervals will fail
- **Returns** –
- **bool** – If the dictionaries contain identical values with uncertainties, and if those values have overlapping confidence intervals

Raises **TypeError** – If `other` is not of the same class as this class nor a subclass of this class

compareInfData(*other*, *sigma*)
Return `True` if contents of `infExp` and `infUnc` agree

Parameters

- **other** (*HomogUniv*) – Object from which to grab group constant dictionaries
- **sigma** (*int*) – Size of confidence interval to apply to quantities with uncertainties. Quantities that do not have overlapping confidence intervals will fail
- **Returns** –
- **bool** – If the dictionaries contain identical values with uncertainties, and if those values have overlapping confidence intervals

Raises **TypeError** – If `other` is not of the same class as this class nor a subclass of this class

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 if `uncertainty`

Raises

- **TypeError** – If the uncertainty flag is not boolean
- **KeyError** – If the variable requested is not stored on the object

See also:

__getitem__()

hasData()
Return True if data is stored on the object.

plot(*qtys*, *limitE=True*, *ax=None*, *logx=None*, *logy=None*, *loglog=None*, *sigma=3*, *xlabel=None*, *ylabel=None*, *legend=None*, *ncol=1*, *steps=True*, *labelFmt=None*, *labels=None*)
Plot homogenized data as a function of energy.

Parameters

- **qtys** (*str or iterable*) – Plot this or these value against energy.
- **limitE** (*bool*) – If given, set the maximum energy value to be that of the micro group structure. By default, SERPENT macro group structures can reach 1E37, leading for a very large tail on the plots.
- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **labels** (*None or iterable*) – Labels to apply to each line drawn. This can be used to identify which bin is plotted as what line.
- **logx** (*bool*) – Apply a log scale to x axis.

- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **sigma** (*int*) – Confidence interval to apply to errors. If not given or `0`, no errors will be drawn.
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **legend** (*bool or str or None*) – Automatically label the plot. No legend will be made if a single item is plotted. Pass one of the following values to place the legend outside the plot: above, right
- **ncol** (*int*) – Integer number of columns to apply to the legend.
- **steps** (*bool*) – If `True`, plot values as constant within energy bins.
- **labelFmt** (*str, optional*) – Formattable string for labeling the individual plots.

String	Replaced value
`{k}`	Name of variable plotted
`{u}`	Name of this universe
`{b}`	Value of burnup in MWd/kgU
`{d}`	Value of burnup in days
`{i}`	Burnup index

Returns Ax on which the data was plotted.

Return type `matplotlib.axes.Axes`

See also:

- `serpentTools.objects.containers.HomogUniv.get()`

6.2.4 serpentTools.objects.XSData

class serpentTools.objects.**XSData**(*name*, *metadata*, *isIso=False*)

Base class for storing cross section data an xsplot file

Parameters

- **name** (*str*) – Name of this material
- **metadata** (*dict*) – Dictionary with file metadata

isIso

Whether this describes individual isotope XS, or whole-material XS

Type bool

MT

Macroscopic cross section integers

Type list

MTdescip
Descriptions of reactions in `MT`

Type list

xsdata

Type `numpy.ndarray`

hasNuData
True if nu data is present

Type bool

metadata
file-wide metadata from the reader

Type dict

hasExpectedData()
Check that the expected data (MT numbers, an energy grid, etc) were collected.

static negativeMTDescription(*mt*)
Gives descriptions for negative MT numbers used by Serpent for whole materials, for neutrons only.

plot(*mts='all'*, *ax=None*, *loglog=False*, *xlabel=None*, *ylabel=None*, *logx=True*, *logy=False*, *title=None*, *legend=None*, *ncol=1*, ***kwargs*)
Return a matplotlib figure for plotting XS.

mts should be a list of the desired MT numbers to plot for this XS. Units should automatically be fixed between micro and macro XS.

Parameters

- **mts** (*int, string, or list of ints*) – If it's a string, it should be 'all'. A single int indicates one MT reaction number. A list should be a list of MT numbers to plot.
- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **loglog** (*bool*) – Apply a log scale to both axes.
- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **title** (*str*) – Title to apply to the figure.
- **legend** (*bool or str or None*) – Automatically label the plot. No legend will be made if a single item is plotted. Pass one of the following values to place the legend outside the plot: above, right
- **ncol** (*int*) – Integer number of columns to apply to the legend.
- **kwargs** (*dict, optional*) – Addition keyword arguments to pass to `matplotlib.pyplot.plot()`

Returns Ax on which the data was plotted.

Return type matplotlib.axes.Axes

Raises **TypeError** – if MT numbers that don't make sense come up

setData(*chunk*)
Parse data from chunk to np array.

setMTs(*chunk*)
Parse chunk to MT numbers and descriptions

setNuData(*chunk*)
Add fission neutrons per fission data

showMT(*retstring=False*)
Pretty prints MT values available for this XS and descriptions.

Parameters **retstring** (*bool*) – return a string if true. Otherwise, print it

tabulate(*mts='all'*, *colnames=None*)
Returns a pandas table, for pretty tabulation in Jupyter notebooks.

Parameters

- **mts** (*int, string, or list of ints*) – If it's a string, it should be 'all', which is default. A single int indicates one MT reaction number. A list should be a list of MT numbers to plot.
- **colnames** (*any type with string representation*) – Column names for each MT number, if you'd like to change them.

Returns

Return type Pandas Dataframe version of the XS data.

Raises

- **ImportError** – If pandas is not installed
- **TypeError** – if MT numbers that don't make sense come up

6.3 Branching Collector

The xs module provides classes that condense *HomogUniv* data read in the *serpentTools.BranchingReader*. The goal of of these is to arrange the data in a structure that is more beneficial for nodal diffusion codes.

BranchCollector	Main class that collects and arranges branched data
BranchedUniv	Class for storing cross sections for a single universe across branches

6.3.1 serpentTools.xs.BranchCollector

class serpentTools.xs.**BranchCollector**(*source*)

Main class that collects and arranges branched data

New in version 0.7.0.

Parameters **source** (str or BranchingReader) – Coefficient file to be read, or a BranchingReader that has read the file of interest

filePath

Location of the read file

Type str

univIndex

Ordered tuple of universe as they appear in the first dimension of all arrays in *xsTables*

Type tuple

universes

Dictionary of universe-specific cross sections. Each entry is a *BranchedUniv* object that stores cross sections for a single universe.

Type dict

xsTables

Dictionary of {k: x} pairs where k corresponds to all cross sections processed and x are large multi-dimensional cross sections. The structure is described with *axis*

Type dict

property axis

Tuple describing axis of underlying data

Each index contains a description of the changes in group constant data along that axis. Can be set to any iterable, but is converted to a tuple to prevent in-place changes, such as appending to a list or removing one item. Passing an ordered object, list, tuple, or numpy.array is preferred, as the conversion to tuple can sort values in un-ordered objects like set or dict strangely.

Examples

```
>>> col.axis
("Universe", "BOR", "TFU", "Burnup", "Group")
>>> infTot = col.xsTables['infTot']
>>> infTot.shape
(5, 3, 3, 3, 2)
# five universes, three BOR perturbations
# three TFU perturbations, three burnups,
# two energy groups
>>> col.axis = ['u', 'b', 't', 'bu', 'g']
>>> col.axis
('u', 'b', 't', 'bu', 'g')
# pass an unordered set
>>> col.axis = {'u', 'b', 't', 'bu', 'g'}
>>> col.axis
('bu', 'u', 't', 'g', 'b')
# incorrectly set axis
>>> col.axis = [1, 2, 3, 4]
ValueError("Current axis has 5 dimensions, not 4")
```

property burnups
Vector of burnups from coefficient file

Can be set to any iterable that has same number of entries as existing burnup. Automatically converts to `numpy.array`

Examples

```
>>> col.burnups
array([0., 1., 10.])
>>> col.burnups = array([0., 5.6, 56.])
>>> col.burnups
array([0., 5.6, 56.])
>>> col.burnups = [0, 1, 10]
# converted to array of integers
>>> col.burnups
array([0, 1, 10])
>>> col.burnups = [0, 1, 2, 3]  # not allowed
ValueError("Current burnup vector has 3 items, not 3")
```

collect (*perturbations=None*)
Parse the contents of the file and collect cross sections

Parameters **perturbations** (*tuple or None*) – Tuple where each entry is a state that is perturbed across the analysis, e.g. `("Tfuel", "RhoCool", "CR")`. These must appear in the same order as they are ordered in the coefficient file. If `None`, then the number of perturbations will be determined from the coefficient file. This is used to set `pertubations` and can be adjusted later

classmethod fromFile (*filePath*, *perturbations=None*)
Create a *BranchCollector* from the contents of the file

Parameters

- **filePath** (*str*) – Location of coefficient file to be read
- **perturbations** (*None or iterable*) – Ordering of perturbation types in coefficient file. If `None`, the number of perturbations will be inferred from file. Otherwise, the number of perturbations must match those in the file. This value can be changed after the fact using *perturbations*, with insight gained from *states*

Returns

- *BranchCollector* object that has processed the contents
- *of the file.*

property perturbations
Iterable indicating the specific perturbation types

Can be set to any iterable, so long as the number of perturbations is preserved. Ordering is important, as changing this does not change the structure of any group constants stored

Example

```
>>> print(col.perturbations)
('BOR', 'TFU')
>>> col.perturbations = ['B', 'T']  # allowed
>>> col.perturbations = [
>>>     'boron conc', 'fuel temp', 'ctrl pos',  # not allowed
>>> ]
ValueError("Current number of perturbations is 2, not 3")
```

property states

Iterable describing each perturbation branch.

Length is equal to that of *perturbations*, and the `i`-th index of `states` indicates the values perturbation `perturbations[i]` experiences.

Can be set to any iterable such that the total number of perturbations is preserved

Examples

```
>>> col.states
(('B1000', 'B750', 'nom'), ('FT1200', 'FT600', 'nom'))
# set as numpy array
>>> states = numpy.array([
    [1000., 750., 0.],
    [1200., 600., 900.]
])
>>> col.states = states
>>> col.states
array([[1000.,  750.,    0],
       [1200.,  600.,  900]])
# set as individual numpy vectors
>>> col.states = (states[0], states[1])
>>> col.states
(array([1000., 750., 0.,]), array([1200., 600., 900.,]))
# pass incorrect shape
>>> col.states = (
>>>     (1000, 750, 0), (1200, 600, 900), (0, 1)
>>> )
ValueError("Current number of perturbations is 2, not 3")
# pass incorrect states for one perturbations
>>> cols.states = (
>>>      (1000, 750, 500, 0), (1200, 600, 900)
>>> )
ValueError("Current number of perturbations for state BOR "
           "is 3, not 4")
```

6.3.2 serpentTools.xs.BranchedUniv

class serpentTools.xs.**BranchedUniv**(*univID*, *collector*, *ndims=None*)

Class for storing cross sections for a single universe across branches

New in version 0.7.0.

Parameters

- **univID** (*str or int*) – Unique ID for this universe
- **collector** (*BranchCollector*) – Class that parses and places branched coefficient data
- **ndims** (*int or iterable*) – Number of perturbation dimensions

filePath

Location of the file that stores the data on this object

Type str

univID

Unique ID for this universe

Type str or int

collector

Class that parses and places branched coefficient data

Type *BranchCollector*

xsTables

Dictionary with keys representing specific values, e.g. `'infTot'` and `'b1Diffcoeff'`. Corresponding values are `BranchedDataTable` objects that store cross section and group constant data across perturbation states

Type dict

__getitem__(*key*)

Access the xsTables dictionary

property axis

Tuple describing axis of underlying data

Note: When setting, the universe index of the axis should not be changed. The changes are passed on to *BranchCollector.axis* with an indicator for universe placed in the correct spot

---

Examples

```
>>> col.axis
('Universe', 'BOR', 'TFU', 'Burnup', 'Group')
>>> u0 = col.universes[0]
>>> u0.axis == col.axis[1:]
True
>>> u0.axis = ['boron conc', 'fuel temp', 'burnup', 'group']
>>> u0.axis
('boron conc', 'fuel temp', 'burnup', 'group')
>>> col.axis
('Universe', 'boron conc', 'fuel temp', 'burnup', 'group')
```

See also:

BranchCollector.axis

property burnups
Vector of burnups from coefficient file

See also:

BranchCollector.burnups

items()
Iterate over names of cross sections and associated objects

property perturbations
Iterable indicating the specific perturbation types

See also:

BranchCollector.perturbations

property states
Iterable describing the names or values of each perturbation branch. Length is equal to that of *perturbations*, and the `i`-th index of `states` indicates the values perturbation `perturbations[i]` experiences.

See also:

BranchCollector.states

CHAPTER

SEVEN

SAMPLERS

The `samplers` module contains classes for reading multiple files of the same type, and averaging quantities across all files. This is often used to remove or reduce the stochastic noise associated with a Monte Carlo program. When generating input files for such runs, it is key that each have a unique random seed. This can be facilitated with the *`seedFiles()`* function, or through the command line with *Random Seed Generation*.

`DetectorSampler`	Class responsible for reading multiple detector files
`SampledDetector`	Class to store aggregated detector data
`DepletionSampler`	Class that reads and stores data from multiple `*dep.m` files
`SampledDepletedMaterial`	Class that stores data from a variety of depleted materials

7.1 serpentTools.samplers.DetectorSampler

class `serpentTools.samplers.`**DetectorSampler**(*files*)

Class responsible for reading multiple detector files

The following checks are performed to ensure that all detectors are of similar structure and content

1. Each parser must have the same detectors
2. The reshaped tally data must be of the same size for all detectors

These tests can be skipped by settings `<sampler.skipPrecheck>` to be `False`.

Parameters **`files`** (*`str or iterable`*) – Single file or iterable (list) of files from which to read. Supports file globs, `*det0.m` expands to all files that end with `det0.m`

`detectors`

Dictionary of key, values pairs for detector names and corresponding *`SampledDetector`* instances

Type dict

`files`

Unordered set containing full paths of unique files read

Type set

`settings`

Dictionary of sampler-wide settings

Type dict

parsers
Unordered set of all parsers that were successful

Type set

map
Dictionary where key, value pairs are files and their corresponding parsers

Type dict

__getitem__(*name*)
Retrieve a detector from *detectors*.

7.2 serpentTools.samplers.SampledDetector

class serpentTools.samplers.**SampledDetector**(*name*, *allTallies*, *allErrors*, *indexes=None*, *grids=None*)

Class to store aggregated detector data

Parameters

- **name** (*str*) – Name of the detector to be sampled
- **allTallies** (*numpy.ndarray or iterable of arrays*) – Array of tally data for each individual detector
- **allErrors** (*numpy.ndarray or iterable of arrays*) – Array of absolute tally errors for individual detectors
- **indexes** (*iterable of string, optional*) – Iterable naming the bins that correspond to reshaped `tally` and *errors*.
- **grids** (*dict, optional*) – Additional grid information, like spatial or energy-wise grid information.

name
Name of this detector

Type str

tallies
Average of tallies from all detectors

Type numpy.ndarray

errors
Uncertainty on *tallies* after propagating uncertainty from all individual detectors

Type numpy.ndarray

deviation
Deviation across all tallies

Type numpy.ndarray

allTallies
Array of tally data from sampled detectors. First dimension is the file index `i`, followed by the tally array for detector `i`.

Type numpy.ndarray

allErrors
Array of uncertainties for sampled detectors. Structure is identical to *allTallies*

Type numpy.ndarray

grids
Dictionary of additional grid information

Type dict or None

indexes
Iterable naming the bins that correspond to reshaped `tally` and *errors*. The tuple `(energy, ymesh, xmesh)` indicates that *tallies* should have three dimensions corresponding to various energy, y-position, and x-position bins. Must be set after *tallies* or *errors* and agree with the shape of each

Type tuple or None

See also:

fromDetectors()

classmethod fromDetectors(*name*, *detectors*)
Create a *SampledDetector* from similar detectors

Parameters

- **name** (*str*) – Name of this detector
- **detectors** (iterable of *serpentTools.Detector*) – Iterable that contains detectors to be averaged. These should be structured identically, in shape of the tally data and the underlying grids and indexes.

Returns

Return type *SampledDetector*

Raises

- **TypeError** – If something other than a *serpentTools.Detector* is found
- **ValueError** – If tally data are not shaped consistently
- **KeyError** – If some grid or index information is missing
- **AttributeError** – If one detector is missing grids entirely but grids are present on other grids

spreadPlot(*xdim=None*, *fixed=None*, *sampleKwargs=None*, *meanKwargs=None*, *ax=None*, *xlabel=None*, *ylabel=None*, *logx=False*, *logy=False*, *loglog=False*, *legend=True*)
Plot the mean tally value against all sampled detector data.

Parameters

- **xdim** (*str*) – Bin index to place on the x-axis
- **fixed** (*None or dict*) – Dictionary controlling the reduction in data down to one dimension
- **sampleKwargs** (*dict, optional*) – Additional matplotlib-acceptable arguments to be passed into the plot when plotting data from unique runs, e.g. `{"c": k, "alpha": 0.5}`.
- **meanKwargs** (*dict, optional*) – Additional matplotlib-acceptable argumentst to be used when plotting the mean value, e.g. `{"c": "b", "marker": "o"}`
- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot

- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **legend** (*bool or str or None*) – Automatically label the plot. No legend will be made if a single item is plotted. Pass one of the following values to place the legend outside the plot: above, right

Returns Ax on which the data was plotted.

Return type `matplotlib.axes.Axes`

7.3 serpentTools.samplers.DepletionSampler

class serpentTools.samplers.**DepletionSampler**(*files*)

Class that reads and stores data from multiple `*dep.m` files

The following checks are performed in order to ensure that depletion files are of similar form:

1. Keys of `materials` dictionary are consistent for all parsers
2. Metadata keys are consistent for all parsers
3. Isotope names and ZZAAA metadata are equal for all parsers

These tests can be skipped by settings `<sampler.skipPrecheck>` to be `False`.

Parameters **files** (*str or iterable*) – Single file or iterable (list) of files from which to read. Supports file globs, `*dep.m` expands to all files that end with `dep.m`

materials

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

Type dict

metadata

Dictionary with file-wide data names as keys and the corresponding data, e.g. `'zai'`: [list of zai numbers]

Type dict

metadataUncs

Dictionary containing uncertainties in file-wide metadata, such as burnup schedule

Type dict

allMdata

Dictionary where key, value pairs are name of metadata and metadata arrays for all runs. Arrays with be of one greater dimension, as the first index corresponds to the file index.

Type dict

files
Unordered set containing full paths of unique files read

Type set

settings
Dictionary of sampler-wide settings

Type dict

parsers
Unordered set of all parsers that were successful

Type set

map
Dictionary where key, value pairs are files and their corresponding parsers

Type dict

__getitem__(*name*)
Retrieve a material from *materials*.

iterMaterials()
Yields material names and objects

7.4 serpentTools.samplers.SampledDepletedMaterial

class serpentTools.samplers.**SampledDepletedMaterial**(*N*, *name*, *metadata*)
Class that stores data from a variety of depleted materials

While `adens`, `mdens`, and `burnup` are accessible directly with `material.adens`, all variables read in from the file can be accessed through the `data` dictionary:

```
>>> assert material.adens is material.data['adens']
>>> assert material.adens is material['adens']
# The three methods are equivalent
```

Note: `free()` sets `allData` to an empty dictionary If `<sampler.freeAll>` is True, then `free` will be called after all files have been read and processed.

Parameters

- **N** (*int*) – Number of containers to expect
- **name** (*str*) – Name of this material
- **metadata** (*dict*) – File-wide metadata for this run. Should contain ZAI and names for all isotopes, days, and burnup schedule

data
dictionary that stores all variable data

Type dict

zai
Isotopic ZZAAA identifiers, e.g. 93325

Changed in version 0.5.1: Now a list of integers, not strings

Type list

names
Names of isotopes, e.g. U235

Type list

days
Vector of total, cumulative days of burnup for the run that created this material

Type numpy.ndarray

burnup
Vector of total, cumulative burnup [MWd/kgU] for this specific material

Type numpy.ndarray

adens
2D array of atomic densities where where row `j` corresponds to isotope `j` and column `i` corresponds to time `i`

Type numpy.ndarray

mdens
2D array of mass densities where where row `j` corresponds to isotope `j` and column `i` corresponds to time `i`

Type numpy.ndarray

uncertainties
Absolute uncertainties for all variables stored in `data`

Type dict

allData
Dictionary where key, value pairs correspond to names of variables stored on this object and arrays of data from all files. The dimensionality will be increased by one, as the first index corresponds to the order in which files were loaded

Type dict

free()
Clear up data from all sampled parsers

plot(*xUnits*, *yUnits*, *timePoints=None*, *names=None*, *ax=None*, *sigma=3*, *xlabel=None*, *ylabel=None*, *logx=False*, *logy=False*, *loglog=False*, *legend=None*, *ncol=1*, *labelFmt=None*, ***kwargs*)
Plot the average of some data vs. time for some or all isotopes.

Note: `kwargs` will be passed to the errorbar plot for all isotopes. If `c='r'` is passed, to make a plot red, then data for all isotopes plotted will be red and potentially very confusing.

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** (matplotlib.axes.Axes, optional) – Ax on which to plot the data. If not provided, create a new plot
- **sigma** (*int*) – Confidence interval to apply to errors. If not given or 0, no errors will be drawn.
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as None, a label will be determined from other arguments. If not None but evaluates to False, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as None, a label will be determined from other arguments. If not None but evaluates to False, do not label.
- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **legend** (*bool or str or None*) – Automatically label the plot. No legend will be made if a single item is plotted. Pass one of the following values to place the legend outside the plot: above, right
- **ncol** (*int*) – Integer number of columns to apply to the legend.
- **labelFmt** (*str, optional*) – Formattable string for labeling the individual plots. If not given, just label as isotope name, e.g. 'U235'. Will make the following substitutions on the labelFmt string, if given:

Keyword	Replacement
'mat'	name of this material
'iso'	specific isotope name
'zai'	specific isotope ZZAAAI

- **kwargs** (*dict, optional*) – Addition keyword arguments to pass to matplotlib.pyplot.errorbar()

Returns Ax on which the data was plotted.

Return type matplotlib.axes.Axes

See also:

- getValues()
- matplotlib.pyplot.errorbar()

spreadPlot(*xUnits*, *yUnits*, *isotope=None*, *zai=None*, *sampleKwargs=None*, *meanKwargs=None*, *timePoints=None*, *ax=None*, *xlabel=None*, *ylabel=None*, *logx=False*, *logy=False*, *loglog=False*, *legend=True*)

Plot the mean quantity and data from all sampled files.

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'
- **isotope** (*str, optional*) – Plot data for this isotope
- **zai** (*int, optional*) – Plot data for this isotope. Not allowed if isotope given.

- **sampleKwargs** (*dict, optional*) – Additional matplotlib-acceptable arguments to be passed into the plot when plotting data from unique runs, e.g. `{"c": k, "alpha": 0.5}`.
- **meanKwargs** (*dict, optional*) – Additional matplotlib-acceptable argumentst to be used when plotting the mean value, e.g. `{"c": "b", "marker": "o"}`
- **timePoints** (*list or None*) – If given, select the time points according to those specified here. Otherwise, select all points
- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **legend** (*bool or str or None*) – Automatically label the plot. No legend will be made if a single item is plotted. Pass one of the following values to place the legend outside the plot: above, right

Returns Ax on which the data was plotted.

Return type `matplotlib.axes.Axes`

CHAPTER

EIGHT

SETTINGS

Presented here are the various settings used by the readers to control what data is stored and how the data is stored. There are specific settings for each reader, and those are of the form `<leader>.<setting>`, so *depletion.materials* is used only by the *DepletionReader*. Without modifying any settings, each reader defaults to storing all of the data present in the output file. Many of the settings instruct the readers to store specific data, such as only materials present in *depletion.materials*.

8.1 Setting Control

Setting control is done using the *rc* global. It acts largely like a dictionary with some validation.

class serpentTools.settings.**rc**

__getitem__()
x.__getitem__(y) <==> x[y]

classmethod __setitem__(*name*, *value*)
Set the value of a specific setting.

Parameters

- **name** (*str*) – Full name of the setting
- **value** (*str*) – 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

classmethod __enter__()
Use as a context manager to easily reset settings

Examples

```
>>> rc["serpentVersion"] = "2.1.30"
>>> rc["serpentVersion"]
"2.1.30"
>>> with rc:
...     rc["serpentVersion"] = "2.1.29"
...     print(rc["serpentVersion"])
"2.1.29"
>>> rc["serpentVersion"]
"2.1.30"
```

The settings are explained below and are used by the readers to control what data are stored when reading from files.

8.2 Shared Settings

There are some settings that are shared between some or all readers. Any setting that does not have a `.` applies to all readers, such as *verbosity*. Below are a list of setting leaders and the readers impacted by this setting:

Leader	
`xs`	*`ResultsReader`* and *`BranchingReader`*

Here are links to locations in the Examples that display how the settings can be used on the various readers.

Reader	Link to example
`BranchingReader`	*User Control*
`DepletionReader`	*Settings*

8.3 Default Settings

8.3.1 `branching.floatVariables`

Names of state data variables to convert to floats for each branch:

```
Default: []
Type: list
```

8.3.2 `branching.intVariables`

Name of state data variables to convert to integers for each branch:

```
Default: []
Type: list
```

8.3.3 depletion.materialVariables

Names of variables to store. Empty list -> all variables:

```
Default: []
Type: list
```

8.3.4 depletion.materials

Names of materials to store. Empty list -> all materials:

```
Default: []
Type: list
```

8.3.5 depletion.metadataKeys

Non-material data to store, i.e. zai, isotope names, burnup schedule, etc:

```
Default: ['ZAI', 'NAMES', 'DAYS', 'BU']
Type: list
Options: [ZAI, NAMES, DAYS, BU]
```

8.3.6 depletion.processTotal

Option to store the depletion data from the TOT block:

```
Default: True
Type: bool
```

8.3.7 detector.names

List of detectors to store. Empty list -> store all detectors:

```
Default: []
Type: list
```

8.3.8 microxs.getFY

If true, store the fission yields:

```
Default: True
Type: bool
```

8.3.9 microxs.getFlx

If true, store the group flux ratios:

```
Default: True
Type: bool
```

8.3.10 microxs.getXS

If true, store the micro-group cross sections:

```
Default: True
Type: bool
```

8.3.11 sampler.allExist

True if all the files should exist. Suppresses errors if a file does not exist:

```
Default: True
Type: bool
```

8.3.12 sampler.freeAll

If true, do not retain data from parsers after reading. Limits memory usage after reading:

```
Default: False
Type: bool
```

8.3.13 sampler.raiseErrors

If True, stop at the first error. Otherwise, continue reading but make a note about the error:

```
Default: True
Type: bool
```

8.3.14 sampler.skipPrecheck

If True, no checks are performed prior to preparing data. Set this to be True only if you know all files contain the same data as errors may arise:

```
Default: False
Type: bool
```

8.3.15 serpentVersion

Version of Serpent:

```
Default: 2.1.31
Type: str
Options: [2.1.29, 2.1.30, 2.1.31]
```

8.3.16 verbosity

Set the level of errors to be shown:

```
Default: warning
Type: str
Options: [critical, error, warning, info, debug]
```

8.3.17 xs.getB1XS

If true, store the critical leakage cross sections:

```
Default: True
Type: bool
```

8.3.18 xs.getInfXS

If true, store the infinite medium cross sections:

```
Default: True
Type: bool
```

8.3.19 xs.reshapeScatter

If true, reshape the scattering matrices to square matrices. By default, these matrices are stored as vectors:

```
Default: False
Type: bool
```

8.3.20 xs.variableExtras

Full SERPENT name of variables to be read:

```
Default: []
Type: list
```

8.3.21 xs.variableGroups

Name of variable groups from variables.yaml to be expanded into SERPENT variable to be store:

```
Default: []
Type: list
```

CHAPTER

NINE

MISCELLANEOUS

9.1 Data Module

The `data` module exists to assist with reproducing example problems and the test interface. It should be used sparsely unless following along with an example as it is subject to change.

readDataFile	Return a reader for the data file (test or example) file.
getFile	Retrieve the path to one of the reference or example files

9.1.1 serpentTools.data.readDataFile

`serpentTools.data.`**`readDataFile`**(*path*, ***kwargs*)

Return a reader for the data file (test or example) file.

All additional keyword arguments will be passed to *serpentTools.read()*.

Parameters **path** (*str*) – The name of the file without any additional directory information

Returns The reader that has processed this file and stored all the data

Return type object

:raises IOError:: If the file for `path` does not exist

Examples

Obtain a detector file from the examples

```
>>> import serpentTools
>>> d = serpentTools.readDataFile('bwr_det0.m')
>>> d.detectors.keys()
dict_keys(['spectrum', 'xymesh'])
```

See also:

- *serpentTools.read()*

9.1.2 serpentTools.data.getFile

serpentTools.data.**getFile**(*path*)
Retrieve the path to one of the reference or example files

Parameters **path** (*str*) – The name of the file without any additional directory information

Returns Path to a data file that can be read with one of the readers or with *serpentTools.read()*

Return type str

:raises IOError:: If no match for path exists

9.2 Exceptions

serpentTools relies on a few custom exceptions, documented below for convenience.

Note: Developers should rely on built-in exceptions whenever possible. These will be fading out over coming versions.

SerpentToolsException	Base-class for all exceptions in this project
SamplerError	Base-class for errors in sampling process
MismatchedContainersError	Attempting to sample from dissimilar containers

9.2.1 serpentTools.messages.SerpentToolsException

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

9.2.2 serpentTools.messages.SamplerError

exception serpentTools.messages.**SamplerError**
Base-class for errors in sampling process

9.2.3 serpentTools.messages.MismatchedContainersError

exception serpentTools.messages.**MismatchedContainersError**
Attempting to sample from dissimilar containers

9.3 Random Seed Module

Exposed through the `serpentTools.seed` module are functions to assist in generating repeated input files with unique random seeds. These are used to support the command line sub-command *Random Seed Generation*.

seedFiles	Copy input file multiple times with unique seeds.
generateSeed	Generate a random seed with `length` digits

9.3.1 serpentTools.seed.seedFiles

serpentTools.seed.**seedFiles**(*inputFile*, *numSeeds*, *seed=None*, *outputDir=None*, *link=False*, *length=10*)
Copy input file multiple times with unique seeds.

Parameters

- **inputFile** (*str*) – Path to input file
- **numSeeds** (*int*) – Number of files to create
- **seed** (*int*) – Optional argument to set the seed of the builtin random number generator
- **outputDir** (*str*) – Path to desired output directory. Files will be copied here. If the folder does not exist, try to make the directory. Assumes path relative to directory that contains the input file
- **link** (*bool*) – If True, do not copy the full file. Instead, create a new file with `'include <inputFile>'` and the new seed declaration.
- **length** (*int*) – Number of digits for random seeds

Returns List of the names of all files created

Return type list

:raises OSError:: If the requested input file could not be found and `link` does not evaluate to true. :raises ValueError:: If the number of requested seeds is not a positive integer, nor can be converted to one, or if the length of the random seeds cannot be converted to a positive integer. :raises TypeError:: Raised if the values passed to `length` or `nseeds` cannot be converted to integers with `int()`

See also:

generateSeed() `random` `random.seed()` `random.getrandbits()`

9.3.2 serpentTools.seed.generateSeed

serpentTools.seed.**generateSeed**(*length*)

Generate a random seed with `length` digits

Parameters **length** (*int*) – Positive number of digits in seed

Returns Randomly generated seed with `length` digits

Return type int

:raises ValueError:: If the value of `length` is non-positive :raises TypeError:: If `length` is not an integer, nor can be coerced into one with `int()`

CHAPTER

TEN

VARIABLE GROUPS

The *BranchingReader* and *ResultsReader* can also be used to store cross sections and other group constant data on *HomogUniv* objects. The group constants to be stored are controlled by two distinct settings: *xs.variableExtras* and *xs.variableGroups*. The former includes the names of Serpent variables **exactly** how they appear in the specific output file. The latter variables that are commonly grouped together, such as kinetic parameters or general cross sections, without having to exactly specify the names of each variable. Without adjusting these settings, the *BranchingReader* and *ResultsReader* will store all data present in the file.

Below are the various variable groups that can be used in the *xs.variableGroups* setting. These groups follow an inheritance-like pattern. For example, using Serpent 2.1.30, any group not directly placed in the *2.1.30* block, such as `eig`, will be replaced by those variables present in the *base* block.

Below are the variable groups that correspond to each version of Serpent supported by this project. Under each block, such as *six-ff*, are the original `SERPENT_STYLE_VARIABLES` and their corresponding variable names converted to `mixedCaseNames`. This conversion is done to fit the style of the project and reduce a few keystrokes for accessing variable names.

10.1 2.1.31

Variable groups for Serpent 2.1.31 are identical to those from Serpent 2.1.30

10.2 2.1.30

10.2.1 n-balance

- `BALA_LOSS_NEUTRON_CAPT` → `balaLossNeutronCapt`
- `BALA_LOSS_NEUTRON_CUT` → `balaLossNeutronCut`
- `BALA_LOSS_NEUTRON_ERR` → `balaLossNeutronErr`
- `BALA_LOSS_NEUTRON_FISS` → `balaLossNeutronFiss`
- `BALA_LOSS_NEUTRON_LEAK` → `balaLossNeutronLeak`
- `BALA_LOSS_NEUTRON_TOT` → `balaLossNeutronTot`
- `BALA_NEUTRON_DIFF` → `balaNeutronDiff`
- `BALA_SRC_NEUTRON_FISS` → `balaSrcNeutronFiss`
- `BALA_SRC_NEUTRON_NXN` → `balaSrcNeutronNxn`
- `BALA_SRC_NEUTRON_SRC` → `balaSrcNeutronSrc`

- BALA_SRC_NEUTRON_TOT → balaSrcNeutronTot
- BALA_SRC_NEUTRON_VR → balaSrcNeutronVr

10.2.2 optimization

- DOUBLE_INDEXING → doubleIndexing
- MG_MAJORANT_MODE → mgMajorantMode
- OPTIMIZATION_MODE → optimizationMode
- RECONSTRUCT_MACROXS → reconstructMacroxs
- RECONSTRUCT_MICROXS → reconstructMicroxs
- SPECTRUM_COLLAPSE → spectrumCollapse

10.2.3 rad-data

- ACTINIDE_ACTIVITY → actinideActivity
- ACTINIDE_DECAY_HEAT → actinideDecayHeat
- ACTINIDE_ING_TOX → actinideIngTox
- ACTINIDE_INH_TOX → actinideInhTox
- ALPHA_DECAY_SOURCE → alphaDecaySource
- CS134_ACTIVITY → cs134Activity
- ELECTRON_DECAY_SOURCE → electronDecaySource
- FISSION_PRODUCT_ACTIVITY → fissionProductActivity
- FISSION_PRODUCT_DECAY_HEAT → fissionProductDecayHeat
- FISSION_PRODUCT_ING_TOX → fissionProductIngTox
- FISSION_PRODUCT_INH_TOX → fissionProductInhTox
- INGENSTION_TOXICITY → ingenstionToxicity
- INHALATION_TOXICITY → inhalationToxicity
- NEUTRON_DECAY_SOURCE → neutronDecaySource
- PHOTON_DECAY_SOURCE → photonDecaySource
- SR90_ACTIVITY → sr90Activity
- TE132_ACTIVITY → te132Activity
- TOT_ACTIVITY → totActivity
- TOT_DECAY_HEAT → totDecayHeat
- TOT_SF_RATE → totSfRate

10.2.4 six-ff

- SIX_FF_EPSILON → sixFfEpsilon
- SIX_FF_ETA → sixFfEta
- SIX_FF_F → sixFfF
- SIX_FF_KEFF → sixFfKeff
- SIX_FF_KINF → sixFfKinf
- SIX_FF_LF → sixFfLf
- SIX_FF_LT → sixFfLt
- SIX_FF_P → sixFfP

10.3 base

10.3.1 adf

- DF_CORN_AREA → dfCornArea
- DF_CORN_DF → dfCornDf
- DF_CORN_IN_CURR → dfCornInCurr
- DF_CORN_NET_CURR → dfCornNetCurr
- DF_CORN_OUT_CURR → dfCornOutCurr
- DF_HET_CORN_FLUX → dfHetCornFlux
- DF_HET_SURF_FLUX → dfHetSurfFlux
- DF_HET_VOL_FLUX → dfHetVolFlux
- DF_HOM_CORN_FLUX → dfHomCornFlux
- DF_HOM_SURF_FLUX → dfHomSurfFlux
- DF_HOM_VOL_FLUX → dfHomVolFlux
- DF_MID_AREA → dfMidArea
- DF_MID_IN_CURR → dfMidInCurr
- DF_MID_NET_CURR → dfMidNetCurr
- DF_MID_OUT_CURR → dfMidOutCurr
- DF_N_CORN → dfNCorn
- DF_N_SURF → dfNSurf
- DF_SURFACE → dfSurface
- DF_SURF_AREA → dfSurfArea
- DF_SURF_DF → dfSurfDf
- DF_SURF_IN_CURR → dfSurfInCurr
- DF_SURF_NET_CURR → dfSurfNetCurr

- DF_SURF_OUT_CURR → dfSurfOutCurr
- DF_SYM → dfSym
- DF_VOLUME → dfVolume

10.3.2 albedos

- ALB_FLIP_DIR → albFlipDir
- ALB_IN_CURR → albInCurr
- ALB_N_SURF → albNSurf
- ALB_OUT_CURR → albOutCurr
- ALB_PART_ALB → albPartAlb
- ALB_SURFACE → albSurface
- ALB_TOT_ALB → albTotAlb

10.3.3 arr-estimators

- CONVERSION_RATIO → conversionRatio
- PU239_CAPT → pu239Capt
- PU239_FISS → pu239Fiss
- PU240_CAPT → pu240Capt
- PU240_FISS → pu240Fiss
- PU241_CAPT → pu241Capt
- PU241_FISS → pu241Fiss
- SM149_CAPT → sm149Capt
- U235_CAPT → u235Capt
- U235_FISS → u235Fiss
- U238_CAPT → u238Capt
- U238_FISS → u238Fiss
- XE135_CAPT → xe135Capt

10.3.4 burnup-coeff

- BURNUP → burnup
- BURN_DAYS → burnDays
- BURN_MATERIALS → burnMaterials
- BURN_MODE → burnMode
- BURN_STEP → burnStep
- COEF_BRANCH → coefBranch

- COEF_BU_STEP → coefBuStep
- COEF_IDX → coefIdx

10.3.5 diffusion

- CMM_DIFFCOEF → cmmDiffcoef
- CMM_DIFFCOEF_X → cmmDiffcoefX
- CMM_DIFFCOEF_Y → cmmDiffcoefY
- CMM_DIFFCOEF_Z → cmmDiffcoefZ
- CMM_TRANSPXS → cmmTranspxs
- CMM_TRANSPXS_X → cmmTranspxsX
- CMM_TRANSPXS_Y → cmmTranspxsY
- CMM_TRANSPXS_Z → cmmTranspxsZ

10.3.6 eig

- ABS_KEFF → absKeff
- ABS_KINF → absKinf
- ANA_KEFF → anaKeff
- COL_KEFF → colKeff
- GEOM_ALBEDO → geomAlbedo
- IMP_KEFF → impKeff

10.3.7 files

- BRA_DATA_DILE_PATH → braDataDilePath
- DECAY_DATA_DILE_PATH → decayDataDilePath
- NFY_DATA_DILE_PATH → nfyDataDilePath
- SFY_DATA_DILE_PATH → sfyDataDilePath
- XS_DATA_FILE_PATH → xsDataFilePath

10.3.8 gc-meta

- GC_UNIVERSE_NAME → gcUniverseName
- INF_FISS_FLX → infFissFlx
- INF_FLX → infFlx
- INF_KINF → infKinf
- INF_MICRO_FLX → infMicroFlx
- MACRO_E → macroE

- MACRO_NG → macroNg
- MICRO_E → microE
- MICRO_NG → microNg

10.3.9 kinetics

- ADJ_IFP_ANA_BETA_EFF → adjIfpAnaBetaEff
- ADJ_IFP_ANA_LAMBDA → adjIfpAnaLambda
- ADJ_IFP_GEN_TIME → adjIfpGenTime
- ADJ_IFP_IMP_BETA_EFF → adjIfpImpBetaEff
- ADJ_IFP_IMP_LAMBDA → adjIfpImpLambda
- ADJ_IFP_LIFETIME → adjIfpLifetime
- ADJ_IFP_ROSSI_ALPHA → adjIfpRossiAlpha
- ADJ_INV_SPD → adjInvSpd
- ADJ_MEULEKAMP_BETA_EFF → adjMeulekampBetaEff
- ADJ_MEULEKAMP_LAMBDA → adjMeulekampLambda
- ADJ_NAUCHI_BETA_EFF → adjNauchiBetaEff
- ADJ_NAUCHI_GEN_TIME → adjNauchiGenTime
- ADJ_NAUCHI_LAMBDA → adjNauchiLambda
- ADJ_NAUCHI_LIFETIME → adjNauchiLifetime
- ADJ_PERT_BETA_EFF → adjPertBetaEff
- ADJ_PERT_GEN_TIME → adjPertGenTime
- ADJ_PERT_LIFETIME → adjPertLifetime
- ADJ_PERT_ROSSI_ALPHA → adjPertRossiAlpha
- BETA_EFF → betaEff
- FWD_ANA_BETA_ZERO → fwdAnaBetaZero
- FWD_ANA_LAMBDA → fwdAnaLambda
- LAMBDA → lambda

10.3.10 lifetime

- ANA_DELAYED_EMTIME → anaDelayedEmtime
- ANA_MEAN_NCOL → anaMeanNcol
- ANA_SLOW_TIME → anaSlowTime
- ANA_THERM_FRAC → anaThermFrac
- ANA_THERM_TIME → anaThermTime

10.3.11 memory

- ALLOC_MEMSIZE → allocMemsize
- AVAIL_MEM → availMem
- MAT_MEMSIZE → matMemsize
- MEMSIZE → memsize
- MISC_MEMSIZE → miscMemsize
- RES_MEMSIZE → resMemsize
- UNKNOWN_MEMSIZE → unknownMemsize
- UNUSED_MEMSIZE → unusedMemsize
- XS_MEMSIZE → xsMemsize

10.3.12 misc

- ANA_AFGE → anaAfge
- ANA_ALF → anaAlf
- ANA_EALF → anaEalf
- FISSE → fisse
- IMP_AFGE → impAfge
- IMP_ALF → impAlf
- IMP_EALF → impEalf
- NUBAR → nubar

10.3.13 n-balance

- BALA_LOSS_NEUTRON_CAPT → balaLossNeutronCapt
- BALA_LOSS_NEUTRON_CUT → balaLossNeutronCut
- BALA_LOSS_NEUTRON_FISS → balaLossNeutronFiss
- BALA_LOSS_NEUTRON_LEAK → balaLossNeutronLeak
- BALA_LOSS_NEUTRON_TOT → balaLossNeutronTot
- BALA_NEUTRON_DIFF → balaNeutronDiff
- BALA_SRC_NEUTRON_FISS → balaSrcNeutronFiss
- BALA_SRC_NEUTRON_NXN → balaSrcNeutronNxn
- BALA_SRC_NEUTRON_SRC → balaSrcNeutronSrc
- BALA_SRC_NEUTRON_TOT → balaSrcNeutronTot
- BALA_SRC_NEUTRON_VR → balaSrcNeutronVr

10.3.14 neutron-physics

- DOPPLER_PREPROCESSOR → dopplerPreprocessor
- IMPL_CAPT → implCapt
- IMPL_FISS → implFiss
- IMPL_NXN → implNxn
- NEUTORN_ERG_NE → neutornErgNe
- NEUTRON_EMAX → neutronEmax
- NEUTRON_EMIN → neutronEmin
- NEUTRON_ERG_TOL → neutronErgTol
- SAMPLE_CAPT → sampleCapt
- SAMPLE_FISS → sampleFiss
- SAMPLE_SCATT → sampleScatt
- TMS_MODE → tmsMode
- USE_DBRC → useDbrc
- USE_DELNU → useDelnu
- USE_URES → useUres

10.3.15 nuclides

- TOT_DECAY_NUCLIDES → totDecayNuclides
- TOT_DOSIMETRY_NUCLIDES → totDosimetryNuclides
- TOT_NUCLIDES → totNuclides
- TOT_PHOTON_NUCLIDES → totPhotonNuclides
- TOT_REA_CHANNELS → totReaChannels
- TOT_TRANSMU_REA → totTransmuRea
- TOT_TRANSPORT_NUCLIDES → totTransportNuclides

10.3.16 optimization

- MG_MAJORANT_MODE → mgMajorantMode
- OPTIMIZATION_MODE → optimizationMode
- RECONSTRUCT_MACROXS → reconstructMacroxs
- RECONSTRUCT_MICROXS → reconstructMicroxs
- SPECTRUM_COLLAPSE → spectrumCollapse

10.3.17 parallel

- MPI_REPRODUCIBILITY → mpiReproducibility
- MPI_TASKS → mpiTasks
- OMP_HISTORY_PROFILE → ompHistoryProfile
- OMP_REPRODUCIBILITY → ompReproducibility
- OMP_THREADS → ompThreads
- SHARE_BUF_ARRAY → shareBufArray
- SHARE_RES2_ARRAY → shareRes2Array

10.3.18 parameters

- B1_BURNUP_CORRECTION → b1BurnupCorrection
- B1_CALCULATION → b1Calculation
- BATCH_INTERVAL → batchInterval
- CYCLES → cycles
- GROUP_CONSTANT_GENERATION → groupConstantGeneration
- IMPLICIT_REACTION_RATES → implicitReactionRates
- NEUTRON_TRANSPORT_MODE → neutronTransportMode
- PHOTON_TRANSPORT_MODE → photonTransportMode
- POP → pop
- SEED → seed
- SKIP → skip
- SRC_NORM_MODE → srcNormMode
- UFS_MODE → ufsMode
- UFS_ORDER → ufsOrder

10.3.19 pin-power

- PPW_FF → ppwFf
- PPW_HOM_FLUX → ppwHomFlux
- PPW_LATTICE → ppwLattice
- PPW_LATTICE_TYPE → ppwLatticeType
- PPW_PINS → ppwPins
- PPW_POW → ppwPow

10.3.20 poisons

- I135_MICRO_ABS → i135MicroAbs
- I135_YIELD → i135Yield
- PM147_MICRO_ABS → pm147MicroAbs
- PM147_YIELD → pm147Yield
- PM148M_MICRO_ABS → pm148mMicroAbs
- PM148M_YIELD → pm148mYield
- PM148_MICRO_ABS → pm148MicroAbs
- PM148_YIELD → pm148Yield
- PM149_MICRO_ABS → pm149MicroAbs
- PM149_YIELD → pm149Yield
- SM149_MACRO_ABS → sm149MacroAbs
- SM149_MICRO_ABS → sm149MicroAbs
- SM149_YIELD → sm149Yield
- XE135_MACRO_ABS → xe135MacroAbs
- XE135_MICRO_ABS → xe135MicroAbs
- XE135_YIELD → xe135Yield

10.3.21 rad-data

- ACTINIDE_ACTIVITY → actinideActivity
- ACTINIDE_DECAY_HEAT → actinideDecayHeat
- ACTINIDE_ING_TOX → actinideIngTox
- ACTINIDE_INH_TOX → actinideInhTox
- ALPHA_DECAY_SOURCE → alphaDecaySource
- BETA_DECAY_SOURCE → betaDecaySource
- CS134_ACTIVITY → cs134Activity
- FISSION_PRODUCT_ACTIVITY → fissionProductActivity
- FISSION_PRODUCT_DECAY_HEAT → fissionProductDecayHeat
- FISSION_PRODUCT_ING_TOX → fissionProductIngTox
- FISSION_PRODUCT_INH_TOX → fissionProductInhTox
- INGENSTION_TOXICITY → ingenstionToxicity
- INHALATION_TOXICITY → inhalationToxicity
- NEUTRON_DECAY_SOURCE → neutronDecaySource
- PHOTON_DECAY_SOURCE → photonDecaySource
- SR90_ACTIVITY → sr90Activity

- TE132_ACTIVITY → te132Activity
- TOT_ACTIVITY → totActivity
- TOT_DECAY_HEAT → totDecayHeat
- TOT_SF_RATE → totSfRate

10.3.22 sampling

- AVG_REAL_COL → avgRealCol
- AVG_SURF_CROSS → avgSurfCross
- AVG_TRACKING_LOOPS → avgTrackingLoops
- AVG_TRACKS → avgTracks
- AVG_VIRT_COL → avgVirtCol
- DT_EFF → dtEff
- DT_FRAC → dtFrac
- DT_THRESH → dtThresh
- LOST_PARTICLES → lostParticles
- MIN_MACROXS → minMacroxs
- REA_SAMPLING_EFF → reaSamplingEff
- REA_SAMPLING_FAIL → reaSamplingFail
- ST_FRAC → stFrac
- TOT_COL_EFF → totColEff

10.3.23 stats

- CYCLE_IDX → cycleIdx
- MEAN_POP_SIZE → meanPopSize
- MEAN_POP_WGT → meanPopWgt
- SIMULATION_COMPLETED → simulationCompleted
- SOURCE_POPULATION → sourcePopulation

10.3.24 times

- BATEMAN_SOLUTION_TIME → batemanSolutionTime
- BURNUP_CYCLE_TIME → burnupCycleTime
- CPU_USAGE → cpuUsage
- ESTIMATED_RUNNING_TIME → estimatedRunningTime
- INIT_TIME → initTime
- MPI_OVERHEAD_TIME → mpiOverheadTime

- OMP_PARALLEL_FRAC → ompParallelFrac
- PROCESS_TIME → processTime
- RUNNING_TIME → runningTime
- TOT_CPU_TIME → totCpuTime
- TRANSPORT_CPU_USAGE → transportCpuUsage
- TRANSPORT_CYCLE_TIME → transportCycleTime

10.3.25 total-rr

- ALBEDO_LEAKRATE → albedoLeakrate
- INI_BURN_FMASS → iniBurnFmass
- INI_FMASS → iniFmass
- TOT_ABSRATE → totAbsrate
- TOT_BURN_FMASS → totBurnFmass
- TOT_CAPTRATE → totCaptrate
- TOT_CUTRATE → totCutrate
- TOT_FISSRATE → totFissrate
- TOT_FLUX → totFlux
- TOT_FMASS → totFmass
- TOT_GENRATE → totGenrate
- TOT_LOSSRATE → totLossrate
- TOT_PHOTON_PRODRATE → totPhotonProdrate
- TOT_POWDENS → totPowdens
- TOT_POWRER → totPowrer
- TOT_RR → totRr
- TOT_SRCRATE → totSrcrate

10.3.26 ures

- URES_AVAIL → uresAvail
- URES_DILU_CUT → uresDiluCut
- URES_EMAX → uresEmax
- URES_EMIN → uresEmin
- URES_SEED → uresSeed

10.3.27 versions

- COMPILE_DATE → compileDate
- COMPLETE_DATE → completeDate
- CONFIDENTIAL_DATA → confidentialData
- CPU_MHZ → cpuMhz
- CPU_TYPE → cpuType
- DEBUG → debug
- HOSTNAME → hostname
- INPUT_FILE_NAME → inputFileName
- START_DATE → startDate
- TITLE → title
- VERSION → version
- WORKING_DIRECTORY → workingDirectory

10.3.28 xs

- ABS → abs
- CAPT → capt
- CHID → chid
- CHIP → chip
- CHIT → chit
- DIFFCOEF → diffcoef
- FISS → fiss
- INVV → invv
- KAPPA → kappa
- NSF → nsf
- NUBAR → nubar
- RABSXS → rabsxs
- REMXS → remxs
- S0 → s0
- S1 → s1
- S2 → s2
- S3 → s3
- S4 → s4
- S5 → s5
- S6 → s6

- S7 → s7
- SCATT0 → scatt0
- SCATT1 → scatt1
- SCATT2 → scatt2
- SCATT3 → scatt3
- SCATT4 → scatt4
- SCATT5 → scatt5
- SCATT6 → scatt6
- SCATT7 → scatt7
- TOT → tot
- TRANSPXS → transpxs

10.3.29 xs-prod

- SCATT2 → scatt2
- SCATTP0 → scattp0
- SCATTP1 → scattp1
- SCATTP3 → scattp3
- SCATTP4 → scattp4
- SCATTP5 → scattp5
- SCATTP6 → scattp6
- SCATTP7 → scattp7
- SP0 → sp0
- SP1 → sp1
- SP2 → sp2
- SP3 → sp3
- SP4 → sp4
- SP5 → sp5
- SP6 → sp6
- SP7 → sp7

CHAPTER

ELEVEN

COMMAND LINE INTERFACE

`serpentTools` includes minor command line functionality. Provided you have installed according to *Installation*, you can access this interface with:

```
$ python -m serpentTools
```

Where the above command is executed from the terminal, neglecting the `$`. More information can be found in *Terminal terminology*.

Note: Outputs here are accurate up to version `0.6.1+12` Please alert the developers if any major differences are found, or a method to automatically update this can be found

Display the available options by passing the `-h` flag:

```
$ python -m serpentTools -h
 usage: serpentTools [-h] [--version] [-v | -q] [-c CONFIG_FILE]
                     {seed,list,to-matlab} ...

 optional arguments:
   -h, --help            show this help message and exit
   --version             show program's version number and exit
   -v                    increase verbosity v: info, vv: debug
   -q                    suppress warnings and errors q: error, qq: critical
   -c CONFIG_FILE, --config-file CONFIG_FILE
                         path to settings file

 sub-commands:
   {seed,list,to-matlab}
                         sub-command help
     seed                copy an input file with unique random number seeds
     list                show the default settings
     to-matlab           convert output file to .mat file
```

11.1 Random Seed Generation

The `seed` subcommand utilizes the `serpentTools.seed` module to produced repeated input files with unique random seeds This is helpful for running repeated analyses to reduce the stochastic nature of Monte Carlo calculations. Options for forcing the number of digits, utilizing the `SERPENT` include command rather than fully copying the file, and new output directories are afforded to the user.

```
$ python -m serpentTools seed -h
usage: serpentTools seed [-h] [--seed SEED] [-l LENGTH]
                         [--output-dir OUTPUT_DIR] [--link]
                         file N

positional arguments:
  file                  input file to copy
  N                     integer number of files to create

optional arguments:
  -h, --help            show this help message and exit
  --seed SEED           Seed to start with the builtin random generator
  -l LENGTH, --length LENGTH
                        Number of digits in random seeds
  --output-dir OUTPUT_DIR
                        Copy files into this directory
  --link                Reference input file with include statement, rather
                        than copying the whole file
```

11.2 Conversion to Binary `.mat` files

Starting in version 0.6.2, `serpentTools` can convert text SERPENT output files and convert them to binary `.mat` files. This relies upon *scipy*, and can be called from the command line as follows:

```
$ python -m serpentTools to-matlab -h
usage: serpentTools to-matlab [-h] [-o OUTPUT] [-a] [--format {4,5}] [-l]
                              [--large] [--oned {col,row}]
                              file

positional arguments:
  file                  output file to read and convert

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        Name of output file to write. If not given, replace
                        extension with .mat
  -a, --append          Append to existing file if found. Otherwise overwrite.
                        Default: False
  --format {4,5}        Format of file to write. 4 for MATLAB 4, 5 for MATLAB
                        5+. Default: 5
  -l, --longNames       Allow variable names up to 63 characters. Otherwise,
                        enforce 31 character names. Default: False
  --large               Don't compress arrays when writing.
  --oned {col,row}      Write 1D arrays are row or column vectors
```

Conversion will exit with no errors if the file is able to be converted, or with the following exit codes:

- `1`: *scipy* not found
- `3`: That file type is not supported at this time.

If you desperately need a file type to be converted, please reach out to the developers on the GH Issue board. Alternatively, if you're feeling ambitious, follow through the *Developer's Guide* for guidelines on adding the feature and submitting a pull request.

CHAPTER

TWELVE

DEVELOPER'S GUIDE

This section serves as a guide for people who would like to add functionality to this project. If that is not your desire, this chapter can be skipped without any loss of comprehension.

12.1 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!

12.1.1 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.

12.1.2 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.

12.1.3 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!

The `develop` branch is the primary branch for this project. All pull requests, except for bugs on public releases, should be compared against this branch. When a pull request is approved and has passed the required checks, it should be squashed and merged into the develop branch. Squashing a branch converts a series of commits into a single commit onto the main branch, creating a tidy git history.

For pull requests into `master`, as in releases, these should simply be merged without squashing. When viewing the `git log` on the `master` or `develop` branches, one is presented only with approved and closed pull requests, not incremental commits that led to a PR being closed.

12.2 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

12.2.1 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.

12.2.2 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.

12.3 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.

12.3.1 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:

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

or:

```
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

Note: For multi-line docstrings, like the longer `depmtx` above, Leave a full blank line between the first line of the docstring, the summary, and the rest of the documentation

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
----------
```

(continues on next page)

(continued from previous page)

```
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.

12.3.2 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.

Converting

These Jupyter notebooks can be converted to `.rst` files for inclusion in the manual with the command `jupyter nbconvert --to=rst <file>`, with:

```
$ jupyter nbconvert --to=rst Notebook.ipynb
```

However, there are some tweaks that should be made so that the documentation renders properly and has helpful links to objects in the project.

The `nbconvert` command will place the following blocks around python code:

```
.. code:: ipython3

    print('hello world!')

.. parsed-literal::

    hello world!
```

When building this documentation on readthedocs, the `ipython3` statement can cause the code not to be rendered. This is summarized here , but it appears that the `ipython3` lexer is not trivially installed and is not found on readthedocs. For now, all these instances of `ipython3` should be removed from the `.rst` version of the notebook so that the wonderful code examples are proudly displayed in our documentation. The above code block should be replaced with:

```
.. code::

    print('hello world!')

.. parsed-literal::

    hello world!
```

It is recommended to use the template in `examples/rstTemplate.tpl` to ease this conversion process. This can be passed to with

```
$ jupyter nbconvert --to=rst --template=rstTemplate.tpl Notebook.ipynb
```

Upon conversion, move the file into the `docs/examples` directory and include the file name in `docs/examples/index.rst`.

Images

Executing `nbconvert` will create a directory containing the images contained in the notebook. When moving the `.rst` version of the notebook into the `docs/examples` folder, make sure that all links to images are correct.

Linking to the API

When referring to python classes, attributes, functions, or methods, it is strongly recommended to utilize python object references. This creates direct links from your text to the object declaration in our api section that allows people to get more detail on whatever you are referencing, powered by the `docstrings` on that object. Two such examples are given below:

- `:py:class:`serpentTools.parsers.depletion.DepletionReader`` becomes `serpentTools.parsers.depletion.DepletionReader`
- `:py:meth:`~serpentTools.objects.materials.DepletedMaterial.plot`` is shortened to `plot()`

12.3.3 Verifying

You worked hard on this documentation, and we want your great work to be properly displayed once completed. In order to reduce the chances of some errors, try running the following from inside the `docs` directory:

```
$ make clean html
```

Navigate to the files created in `_build/html` to ensure that images are loaded properly, code is rendered properly, and the converted notebook looks exactly how you expect it to look.

Warning: If there is an issue with rendering your example, we will likely call upon you to fix these issues.

Note: Building the documentation locally requires `sphinx` and a handful of other packages. Installing these is outside the scope of this guide, partially because the sphinx team has a great guide already. Check this out if you are having issues running the `make clean install` commands from the docs directory.

12.3.4 Adding Objects to API

New reader or container objects should be included in the api section of the documentation, as with any function that the end user may utilize. For documenting these, we utilize the sphinx autodoc features to use the docstrings to automatically document new features. This is most simply done by calling `.. autoclass::` or `.. autofunction::` like:

```
.. autofunction:: serpentTools.plot.plot

.. autoclass:: serpentTools.parsers.results.ResultsReader
```

For new readers, those should be included in their own file, such as `docs/api/myNewReader.rst`, which can be as bare as:

```
My New Reader
=============

.. autoclass:: serpentTools.parsers.new.MyNewReader
```

Be sure to include your new file in `docs/api/index.rst`, or else your file will be left out of the documentation. Proper documentation of the class or function requires thorough and concise documentation of all attributes, class methods, and construction arguments. Follow the above guides, such as *Docstrings*, and this process *should* go smoothly.

12.4 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*

12.4.1 CI Testing

We utilize Travis to perform our continuous integration, or *CI*. Two types of tests are run: unit tests and tests on the example notebooks. The former are used to perform more granular testing over the project, while the latter ensure our example notebooks are runnable.

Unit Tests

Unit tests should cover all public methods and a sufficient set of use cases to reduce the amount of bugs that make it into releases. Unit tests are run using *pytest* from the project directory:

```
$ ls
examples/ LICENSE serpentTools/ tests/ ...
$ pytest
```

If the `pytest-cov` package is installed, you can view the coverage, or how much of the project is touched by tests, with:

```
$ pytest --cov=serpentTools
```

It is always preferable to increase coverage to decreasing coverage, but minor deviations are acceptable. The coverage is not a factor in passing or failing CI, but substantial changes to the test suite should serve a valid purpose if the coverage does decrease.

Notebook Tests

We try to provide a *Jupyter notebook* for each of the main readers in the package. These are converted to be used in our main documentation, and serve as a handy way for new users to discover the features provided with `serpentTools`. In order to ensure that these are valid notebooks as changes are introduced, our *CI* converts these to python scripts and runs them using the test notebooks script. It is beneficial to run this script during major changes to the API, as well as correcting any errors or deprecated/removed features.

12.4.2 Lint

We also recommend using a *linter* to analyze the project for code that might induce errors and does not conform to our style guide - *Coding Style*. This is applied on proposed changes using flake8. Code to be merged into this project should not cause any unit tests to break, nor introduce lint. If you are working on a feature/bug-fix branch, you can compare the lint that would be introduced with

```
$ git diff --unified=0 develop | flake8 --diff
```

Here, `develop` is the intended target branch into which these changes will be merged.

12.5 Version Control

`serpentTools` uses *git* for version control. All the source code can be found at https://github.com/CORE-GATECH-GROUP/serpent-tools. Two main branches are provided: `master` and `develop`, and the git flow branching model is followed. The `master` branch should be considered stable and updated coincident with each release. The `develop` branch is updated with more frequency as features are introduced. This is the main branch of the project, and features should be introduced off of this branch.

`serpentTools` follows the semantic versioning system, where the version number as found in `setup.py`, `serpentTools/__init__.py`, and `docs/conf.py` has the following form: `major.minor.patch`, e.g. `0.8.0` or `1.1.20`. Each of the numbers should be incremented prior to new releases with the following designation:

1. Changes that are not backwards compatible should be denoted by incrementing the major number.
2. Changes that are backwards compatible and introduce sufficient new features to the API should be denoted by incrementing the minor number.
3. Changes that are largely internal and not recognizable by end-users should be denoted by incrementing the patch number

Note: Until a stable 1.0.0 release, the positions are essentially shifted, e.g. the version is `0.major.minor`

12.5.1 Releases

Releases should be done with git tags from the master branch and then pushed to GitHub. Annotated tags should be used and can be created with:

```
git tag -a <version>
git tag -s <version>
git tag -m <msg> <version>
```

Pushing these tags to GitHub creates a new release. If a message is used, the messages should be a brief message describing the changes on this tag. On the release page, a more detail list of changes, such as pull requests and issues closed, should be listed.

Before and after a release, the project version number should be updated in the following places:

1. `setup.py`
2. `serpentTools/__init__.py`
3. `docs/conf.py`

Following a release, the following actions should be performed:

1. Archived and installable python files should be created
2. Archived and installable python files should be uploaded to the python package index
3. The tag should be pushed to GitHub and marked as a release, including information on the changes introduced in this release

12.5.2 Commit Messages

When possible, please provide commit messages that:

- have a initial single summary line (~<50 characters),
- followed by a blank line,
- followed by as detailed of a description as possible wrapped to ~70 characters wide

Helpful and detailed commit messages can make searching for changes easier and accelerate the review process. As an added benefit, if your pull request is a single commit, GitHub will automatically populate the request summary with your commit message!

Other references:

- git documentation on commit messages
- Good example commit message - Tim Pope

12.6 Logging and Reporting

The primary internal logging is performed with a logger named `serpentTools`, that can be obtained using:

```
>>> import logging
>>> logging.getLogger("serpentTools")
```

If you want to see the messages produced by this logger, you have a few options. First, the Python logging system must be configured. This can be done simply with:

```
>>> import logging
>>> logging.basicConfig(format="%(levelname)-s: %(message)-s")
# Display a basic warning
>>> logging.warning("This is a warning")
WARNING: This is a warning
```

To show the internal messages, one can modify the verbosity through the *Settings* interface with:

```
>>> serpentTools.settings.rc["verbosity"] = "debug"
```

where "debug" can be one of "debug", "info", "warning", "error" or "critical".

Alternatively, the level can be adjusting using the python logging module:

```
>>> logging.getLogger("serpentTools").setLevel(logging.DEBUG)
```

12.6.1 Developer Reference

Note: The use of built-in python warning support through the warnings module should be preferred. These functions will be phased out in future versions

This chapter describes the various functions used to convey progress updates or issues to the user. Functions in this module should be used over a general print statement, as this module can be be extended to log messages to a file in the future. In order of increasing severity:

debug	Log a debug message.
info	Log an info message
warning	Log a warning that something could go wrong or should be avoided.
error	Log that something caused an exception but was suppressed.
critical	Log that something has gone horribly wrong.

serpentTools.messages.debug

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

serpentTools.messages.info

serpentTools.messages.**info**(*message*)
Log an info message

serpentTools.messages.warning

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

serpentTools.messages.error

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

serpentTools.messages.critical

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

12.6.2 Decorators

Two decorators are provided in the `messages` module that are used to indicate functions or methods who's behavior will be changing or removed in the future.

deprecated	Decorator that warns that different function should be used instead.
willChange	Decorator that warns that some functionality may change.

serpentTools.messages.deprecated

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

serpentTools.messages.willChange

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

12.6.3 Custom Handlers

class serpentTools.messages.**DictHandler**(*level=0*)
Bases: `logging.Handler`

Handler that stores log messages in a dictionary

logMessages
Dictionary of lists where each key is a log level such as `'DEBUG'` or `'WARNING'`. The list associated with each key contains all messages called under that logging level

Type dict

close()
Tidy up before removing from list of handlers

emit(*record*)
Store the message in the log messages by level.

Does no formatting to the record, simply stores the message in *logMessages* dictionary according to the records `levelname`

Anticipates a `logging.LogRecord` object

flush()
Clear the log messages dictionary

12.7 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()
```

12.8 Plotting

Many of the readers and containers in this project have plot routines to quickly visualize relevant data. The goal for these plots is to quickly provide presentation-worthy plots by default, with options for formatting via additional arguments. This document serves as a guide for writing plot functions that have a cohesive syntax across the project and require little to no knowledge of *matplotlib* to make a fantastic plot.

Note: Plot functions should contain sufficient formatting such that, with minimal user input, processional and publishable plots are produced.

This cohesion is accomplished by decorating plot routines with the `magicPlotDocDecorator()` and by supporting a collection of additional arguments used for formatting the plot. These magic strings are detailed in *Magic Plot Decorator Options* and can be included in any function decorated by `magicPlotDocDecorator()` as `{legend}`.

When appropriate, plot functions should accept a `matplotlib.axes.Axes` argument on which to plot the data. This is useful for subplotting, or for creating a plot and then continuing the plotting inside this function. If not given, the function should create a new `matplotlib.axes.Axes` object. All plotting should be done on this axes object, such as:

```
def plot(ax=None):
    ax = ax or pyplot.axes()
    ax.plot([1,2,3])
    return ax
```

By returning the axes object, users can further apply labels, place the plot into subplots, or more.

Axes for all plots should be labeled without user intervention, but should support user-defined labels. This process can be expedited by utilizing `formatPlot()` as the last stage of the plotting process. This function accepts a variety of keyword arguments that are used to set axis labels or scaling, even placing the legend outside the plot. Taking the simple plot method from above, we can modify this method to accept user arguments for labels, but also have default labels should the user not provide them:

```
def plot(ax=None, xlabel=None, ylabel=None):
    ax = ax or pyplot.axes()
    ax.plot([1, 2, 3])
    xlabel = xlabel or "Default x label"
    ax = formatPlot(ax, xlabel=xlabel,
                    ylabel=ylabel or "Y axis")
    return ax
```

12.8.1 Plotting Utilities

Plot Functions

The `serpentTools.plot` module contains some functions to assist in plotting and for describing plot functions. These are not required for use, but their behavior, and the advice given in *Plotting*, should be followed. This will yield a consistent and flexible plotting environment to the user without having to dive deep into the `matplotlib` framework.

cartMeshPlot	Create a cartesian mesh plot of the data

Continued on next page

Table 3 – continued from previous page

plot	Shortcut plot for plotting series of labeled data onto a plot

serpentTools.plot.cartMeshPlot

serpentTools.plot.**cartMeshPlot**(*data*, *xticks=None*, *yticks=None*, *ax=None*, *cmap=None*, *logColor=False*, *normalizer=None*, *cbarLabel=None*, *thresh=None*, ***kwargs*)

Create a cartesian mesh plot of the data

Parameters

- **data** (*numpy.array*) – 2D array of data to be plotted
- **xticks** (*iterable or None*) –
- **yticks** (*iterable or None*) – Values corresponding to lower x/y boundary of meshes. If not given, treat this as a matrix plot like `matplotlib.pyplot.imshow()`. If given, they should contain one more item than number of elements in their dimension to give dimensions for all meshes.
- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **cmap** (*str, optional*) – Valid Matplotlib colormap to apply to the plot.
- **logColor** (*bool*) – If true, apply a logarithmic coloring
- **normalizer** (callable or `matplotlib.colors.Normalize`) – Custom normalizer for this plot. If an instance of `matplotlib.colors.Normalize`,
- **normalizer** – Custom normalizer for this plot. If an instance of `matplotlib.colors.Normalize`, use directly. Otherwise, assume a callable object and call as `norm = normalizer(data, xticks, yticks)`
- **cbarLabel** (*None or str*) – Label to apply to colorbar
- **{thresh}** –
- **kwargs** (*dict, optional*) – Addition keyword arguments to pass to `matplotlib.pyplot.pcolormesh()` or `matplotlib.pyplot.imshow()` if `xticks` and `yticks` are `None`

Returns Ax on which the data was plotted.

Return type `matplotlib.axes.Axes`

Raises

- **ValueError** – If `logColor` and data contains negative quantities
- **TypeError** – If only one of `xticks` or `yticks` is `None`.

Examples

```
>>> from serpentTools.plot import cartMeshPlot
>>> from numpy import arange
>>> data = arange(100).reshape(10, 10)
>>> x = y = arange(11)
>>> cartMeshPlot(data, x, y, cbarLabel='Demo')
```

```
>>> from serpentTools.plot import cartMeshPlot
>>> from numpy import  eye
>>> data = eye(10)
>>> for indx in range(10):
...     data[indx] *= indx
>>> cartMeshPlot(data, logColor=True)
```

All values less than or equal to zero are excluded from the color normalization. The `logColor` argument works well for plotting sparse matrices, as the zero-valued indices can be identified without obscuring the trends presented in the non-zero data.

Alternatively, one can use the `thresh` argument to set a threshold for plotted data. Any value less than or equal to `thresh` will not be colored, and the colorbar will be updated to reflect this.

```
>>> from serpentTools.plot import cartMeshPlot
>>> from numpy import arange
>>> data = arange(100).reshape(10, 10)
>>> cartMeshPlot(data, thresh=50)
```

See also:

- `matplotlib.pyplot.pcolormesh()`
- `matplotlib.pyplot.imshow()`
- `matplotlib.colors.Normalize`

serpentTools.plot.plot

`serpentTools.plot.`**`plot`**(*xdata*, *plotData*, *ax=None*, *labels=None*, *yerr=None*, ***kwargs*)

Shortcut plot for plotting series of labeled data onto a plot

If `plotData` is an array, it is assumed that each column represents one set of data to be plotted against `xdata`. The same assumption is made for `yerr` if given.

Parameters

- **`xdata`** (*`numpy.array or iterable`*) – Points along the x axis to plot
- **`plotData`** (*`numpy.array or iterable`*) – Data to be plotted
- **`ax`** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **`labels`** (*`None or iterable`*) – Labels to apply to each line drawn. This can be used to identify which bin is plotted as what line.
- **`yerr`** (*`None or numpy.array or iterable`*) – Absolute error for each data point in `plotData`

- **kwargs** (*dict, optional*) – Addition keyword arguments to pass to `matplotlib.pyplot.plot()` or `matplotlib.pyplot.errorbar()`

Returns Ax on which the data was plotted.

Return type `matplotlib.axes.Axes`

Raises **IndexError** – If `yerr` is not `None` and does not match the shape of `plotData`

Plot Formatters

The following functions can be used to tidy up a plot given our wide-range of supported arguments.

formatPlot	Apply a range of formatting options to the plot.
placeLegend	Add a legend to the axes instance.
setAx_xlims	Set the x limits on an Axes object
setAx_ylims	Set the y limits on an Axes object
addColorbar	Quick utility to add a colorbar to an axes object
normalizerFactory	Construct and return a `Normalize` for this data

serpentTools.utils.plot.formatPlot

serpentTools.utils.plot.**formatPlot**(*ax*, *xlabel=None*, *ylabel=None*, *logx=False*, *logy=False*, *loglog=False*, *title=None*, *legend=True*, *legendcols=1*, *axkwargs=None*, *legendkwargs=None*)

Apply a range of formatting options to the plot.

Parameters

- **ax** (`matplotlib.axes.Axes`, optional) – Ax on which to plot the data. If not provided, create a new plot
- **xlabel** (*str or bool, optional*) – Label to apply to the x-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **ylabel** (*str or bool, optional*) – Label to apply to the y-axis. If given as `None`, a label will be determined from other arguments. If not `None` but evaluates to `False`, do not label.
- **loglog** (*bool*) – Apply a log scale to both axes.
- **logx** (*bool*) – Apply a log scale to x axis.
- **logy** (*bool*) – Apply a log scale to y axis.
- **title** (*str, optional*) – Title to apply to this axes object
- **legend** (*bool or str or None*) – Automatically label the plot. No legend will be made if a single item is plotted. Pass one of the following values to place the legend outside the plot: above, right
- **legendcols** (*int, optional*) – Number of columns to apply to the legend, if applicable
- **axkwargs** (*dict, optional*) – Additional keyword arguments to be passed to `matplotlib.axes.Axes.set()`. Values like `xlabel`, `ylabel`, `xscale`, `yscale`, and title will be respected, even though they are also arguments to this function

- **legendkwargs** (*dict, optional*) – Additional keyword arguments to be passed to matplotlib.axes.Axes.legend(). Values like title and ncol will be respected.

Returns Ax on which the data was plotted.

Return type matplotlib.axes.Axes

Raises **TypeError** – If the ax argument is not an instance of matplotlib.pyplot.axes.Axes

serpentTools.utils.plot.placeLegend

serpentTools.utils.plot.**placeLegend**(*ax*, *legend*, *handlesAndLabels=None*, ***kwargs*)

Add a legend to the axes instance.

A legend will be drawn if at least one of the following criteria are met:

1. legend is not one of the supported string legend control arguments that dictate where the legend should be placed
2. More than one item has been plotted

Parameters

- **ax** (matplotlib.axes.Axes, optional) –
- **on which to plot the data. If not provided, create a new** (*Ax*) –
- **plot** –
- **legend** (*bool or str or None*) –
- **label the plot. No legend will be made if a** (*Automatically*) –
- **item is plotted. Pass one of the following values** (*single*) –
- **place the legend outside the plot** (*to*) –
- **handlesAndLabels** (*tuple (legend handles, labels), optional*) – Exact legend handles (graphical element) and labels (string element) to be used in making the legend. If not provided, will be fetched from matplotlib.axes.Axes.get_legend_handles_labels()
- **kwargs** (*dict, optional*) – Additional keyword arguments to be passed to matplotlib.axes.Axes.legend(), if applicable

Returns

- matplotlib.axes.Axes
- *Ax on which the data was plotted.*

serpentTools.utils.plot.setAx_xlims

serpentTools.utils.plot.**setAx_xlims**(*ax*, *xmin*, *xmax*, *pad=10*)

Set the x limits on an Axes object

Parameters

- **ax** (matplotlib.axes.Axes) – Ax to be updated
- **xmin** (*float*) – Current minimum extent of x axis
- **xmax** (*float*) – Current maximum extent of x axis
- **pad** (*float, optional*) – Padding, in percent, to apply to each side of the current min and max

serpentTools.utils.plot.setAx_ylims

serpentTools.utils.plot.**setAx_ylims**(*ax*, *ymin*, *ymax*, *pad=10*)

Set the y limits on an Axes object

Parameters

- **ax** (matplotlib.axes.Axes) – Ax to be updated
- **ymin** (*float*) – Current minimum extent of y axis
- **ymax** (*float*) – Current maximum extent of y axis
- **pad** (*float, optional*) – Padding, in percent, to apply to each side of the current min and max

serpentTools.utils.plot.addColorbar

serpentTools.utils.plot.**addColorbar**(*ax*, *mappable*, *norm=None*, *cbarLabel=None*)

Quick utility to add a colorbar to an axes object

The color bar is placed adjacent to the provided axes argument, rather than in the provided space.

Parameters

- **mappable** (*iterable*) – Collection of meshes, patches, or values that are used to construct the colorbar.
- **norm** (matplotlib.colors.Normalize, optional) – Normalizer for this plot. Can be a subclass like matplotlib.colors.LogNorm
- **cbarLabel** (*str, optional*) – If given, place this as the y-label for the colorbar

Returns The colorbar that was added

Return type matplotlib.colorbar.Colorbar

serpentTools.utils.plot.normalizerFactory

serpentTools.utils.plot.**normalizerFactory**(*data*, *norm*, *logScale*, *xticks*, *yticks*)

Construct and return a Normalize for this data

Parameters

- **data** (numpy.ndarray) – Data to be plotted and normalized
- **norm** (None or callable or matplotlib.colors.Normalize) – If a Normalize object, then use this as the normalizer. If callable, set the normalizer with norm(data, xticks, yticks). If not None, set the normalizer to be based on the min and max of the data
- **logScale** (*bool*) – If this evaluates to true, construct a matplotlib.colors.LogNorm with the minimum set to be the minimum of the positive values.
- **xticks** (numpy.ndarray) –
- **yticks** (numpy.ndarray) – Arrays ideally corresponding to the data. Used with callable *norm* function.

Returns

- matplotlib.colors.Normalize
- or matplotlib.colors.LogNorm
- *or object* – Object used to normalize colormaps against these data

Docstring Formatters for Plots

Many of the plot routines accept common parameters, such as logx for applying a log scale to the x-axis. To reduce the amount of repeated code, the following formatting functions operate on the docstrings of a plot function/method and save a lot of extra code.

See *Magic Plot Decorator Options* for a listing of what is replaced by what when calling serpentTools.plot.magicPlotDocDecorator().

serpentTools.utils.docstrings.**magicPlotDocDecorator**(*f*)

Decorator that replaces a lot magic strings used in plot functions.

Allows docstrings to contain keyword that will be replaced with a valid and proper explanation of the keyword. Keywords must be wrapped in single brackets, i.e. {x}

Magic Plot Decorator Options

Note: These keyword arguments should only be used when they are appropriate for the specific plot method. Colormaps don't really make sense for line plots, but can be very useful for 2D contour and mesh plots. cmap should be a supported argument for most of these plot types, but not for line plots, unless it makes sense to do such.

1. **ax: ax: matplotlib.axes.Axes or None** Ax on which to plot the data.
2. **cmap: cmap: str or None** Valid Matplotlib colormap to apply to the plot.
3. **kwargs: kwargs:** Addition keyword arguments to pass to
4. **labels: labels: None or iterable** Labels to apply to each line drawn. This can be used to identify which bin is plotted as what line.

5. **legend: legend: bool or str** Automatically label the plot. Pass one of the following values to place the legend outside the plot: above, right
6. **loglog: loglog: bool** Apply a log scale to both axes.
7. **logx: logx: bool** Apply a log scale to x axis.
8. **logy: logy: bool** Apply a log scale to y axis.
9. **matLabelFmt: labelFmt: str or None** Formattable string for labeling the individual plots. If not given, just label as isotope name, e.g. `'U235'`. Will make the following substitutions on the `labelFmt` string, if given:

Keyword	Replacement
`'mat'`	name of this material
`'iso'`	specific isotope name
`'zai'`	specific isotope ZZAAAI

10. **ncol: ncol: int** Integer number of columns to apply to the legend.
11. **rax: matplotlib.axes.Axes** Ax on which the data was plotted.
12. **sigma: sigma: int** Confidence interval to apply to errors. If not given or `0`, no errors will be drawn.
13. **title: title: str** Title to apply to the figure.
14. **xlabel: xlabel: str or None** Label for x-axis.
15. **ylabel: yabel: str or None** Label for y-axis.

12.9 Converter Utilites

One goal for this project is to allow users to use `serpentTools` as a pipeline for taking *SERPENT* data to other programs. The first demonstration of this is creating binary `.mat` files that can be read into MATLAB, when a large text file would cause issues, done with the *`MatlabConverter`*.

Classes intending do first look for specific methods for gathering data on each object. These are obscured from the public API as they don't serve a direct purpose besides a simple method by which to translate data from one place to the next. This collection is called inside the primary `convert` method, which is reponsible for gathering and writing the data into the new form.

Keeping with the *`MatlabConverter`* as an example, it looks for `_gather_matlab` method. This method simply places data into a dictionary, but do to the layout of each object, must be reimplemented as a unique method. Due to the simplicity of `scipy.io.savemat()`, the main writing function, *`serpentTools.io.base.MatlabConverter.convert()`*, is rather light. However, some data forms may require more depth to ensure fidelity in the converted data.

There may be some cases where a conversion does not make sense for all data types. Detector objects are the few pieces of data that can have a spatial grid attached, yet this grid varies across detectors. It may be advantageous to implement converters directly as instance methods. The decision will be up to the developer and the review process in that case.

Converters should also make sure, prior to potentially expensive collection steps are taken, that the user is in the best position to execute this operation. As of this writing and 0.6.1, *scipy* is not a requirement for this package, yet it is required for MATLAB conversion. Therefore, the *`MatlabConverter`* implements a check for *scipy* in *`checkImports()`*. The other check method, *`checkContainerReq()`*, is responsible for checking the container has the required data gathering routines, e.g. `_gather_matlab`. These checks are called at creation of the converter object.

class serpentTools.io.base.**MatlabConverter**(*container*, *output*)
Class for assisting in writing container data to MATLAB

Parameters

- **obj** (serpentTools container) – Parser or container to be written to file
- **fileP** (*str or file-like object*) – Name of the file to write

See also:

- serpentTools.io.toMatlab() - high level function for writing to MATLAB that uses this class

checkContainerReq(*container*)
Ensure that data from the container can be collected.

checkImports()
Ensure that *scipy* >= 1.0 is installed.

convert(*reconvert*, *append=True*, *format='5'*, *longNames=True*, *compress=True*, *oned='row'*)
Save contents of object to .mat file

Parameters

- **fileP** (*str or file-like object*) – Name of the file to write
- **reconvert** (*bool*) – If this evaluates to true, convert values back into their original form as they appear in the output file, e.g. MAT_TOTAL_ING_TOX. Otherwise, maintain the mixedCase style, total_ingTox.
- **append** (*bool*) – If true and a file exists under output, append to that file. Otherwise the file will be overwritten
- **format** (*{'5', '4'}*) – Format of file to write. '5' for MATLAB 5 to 7.2, '4' for MATLAB 4
- **longNames** (*bool*) – If true, allow variable names to reach 63 characters, which works with MATLAB 7.6+. Otherwise, maximum length is 31 characters
- **compress** (*bool*) – If true, compress matrices on write
- **oned** (*{'row', 'col'}:*) – Write one-dimensional arrays as row vectors if oned=='row' (default), or column vectors

See also:

scipy.io.savemat()

12.10 Parsing Engines

The *KeywordParser* and *PatternReader* 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
```

(continues on next page)

(continued from previous page)

```
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.

KeywordParser	Class for parsing a file for chunks separated by various keywords.
PatternReader	Class that can read over a file looking for patterns.

12.10.1 serpentTools.engines.KeywordParser

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

Most recently read line

Type str

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.

12.10.2 serpentTools.engines.PatternReader

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
Most recently read line

Type str

match
Match from the most recently read line

Type regular expression match or None

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*

The *CSCStreamProcessor* is provided to help with reading sparse matrices provided by Serpent. These are current found in the depletion and fission matrix files.

class serpentTools.parsers.base.**CSCStreamProcessor**(*stream*, *regex*, *dtype=<class 'float'>*)
Read through a block of text and produce sparse matrices

Note: Rows and columns matched by `regex` will be reduced by one prior to storage. Since we primarily interact with one-indexed MATLAB arrays, we need to convert the indices to something *numpy* can properly understand

---

Parameters

- **stream** (*IO stream from an opened file*) – Object with a `readline` function that returns the next line of text to be read
- **regex** (*str or compiled regular expression*) – Regular expression that matches the following:

0. Row of matrix
1. Column of matrix
2. Values to be added into resulting matrix.

All values in `match.groups()[2:]` will be converted to `datatype` and appended into *data*. The rows and columns are used to populate *indices* and *indptr* vectors

- **datatype** (*object*) – Data type of the numeric values of this matrix.

data
Column matrix of all values matched by `regex` after first two positions. Each columns can be used to build a sparse matrix with *indices* and *indptr*

Type numpy.ndarray

indices
CSC-format indices pointer array

Type numpy.ndarray

indptr
CSC-format index pointer array. Row indices for column `i` are stored in `indices[indptr[i]:indptr[i + 1]]`. Values for column `i` are stored in `data[indptr[i]:intptr[i + 1]]`.

Type numpy.ndarray

line
Last line read after calling `process()`. Will be the first non-empty line that does not match the passed regular expression

Type str

See also:

- scipy.sparse.csc_matrix

12.11 Adding Support for Serpent Versions

One of the main goals for this project is to include support for more `SERPENT` versions. The goal of this document is to detail what steps are required in order to include support for additional `SERPENT` versions.

Note: Supporting additional versions is an ongoing task. See #117 and linked issues for updates

12.11.1 Checklist

1. Include the version as an option in the *serpentVersion* setting - *serpentVersion*
2. Update the *Default Settings*
3. Update `serpentTools/variables.yaml` to include the version and any new variable groups present in the results files - *New Variables*
4. If `serpentTools/variables.yaml` is updated, update *Variable Groups* using the script `docs/rstVariableGroups.py`

12.11.2 New Variables

As detailed in *Variable Groups*, `serpentTools` utilizes a collection of variable groups to expedite what data is extracted from results and branching files. Some versions of `SERPENT` may have different data, as more features are added to the code, whiles some may have the same data with different names. Variables and variable groups can be added to `serpentTools/variables.yaml` as sets. Since the order of the variables present in the file is not important, and we are simply interested in *if* a variable found in a file is in a group, the `set` provides a good framework for this inspection.

The basic structure of `serpentTools/variables.yaml` is given below for a single version:

```
version
    |- group_0
        | - variable_0
        | - variable_1
         ...
    |- group_1
         | - variable_0
         | - variable_1
         ...
    ...
```

There is also a `base` version that serves as a catch-all, as detailed in *Variable Lookup Procedure*.

Variables are loaded from `serpentTools/variables.yaml` as nested dictionaries:

```
variables = {
    version0: {group_0: {variable_0, variable_1, ...},
               group_1: {variable_0, variable_1, ...},
               ...
              },
    version1: {group_0: {variable_0, variable_1, ...},
               group_1: {variable_0, variable_1, ...},
               ...
              },
    ...
    }
```

Variable Lookup Procedure

The lookup procedure for variable groups is as follows. For a specific `SERPENT` version `V` and variable group `g`,

- **If the variable group is explicitly declared under `variables[V]`, then take the** variables from `variables[V][g]`
- **If the group is declared under `variables['base']`, then** take the variables from `variables['base'][g]`
- Otherwise, raise an error

The `'base'` serves to store variables that are invariant across all `SERPENT` versions. As the development of `SERPENT` progresses, it is not unlikely that the `'base'` group may dwindle, as each version listed in `serpentTools/variables.yaml` expands.

Shared Variable Groups

For variable groups that are similar between two versions but not contained in `'base'`, one can take advantage of YAML anchors and aliases, implemented in `PyYAML` as `&` and `*` characters. An example of sharing a group between two versions this way is given below:

```
parentV:
  parentG: !!set &parentG
    {vx0, vx1, ...}
childV:
  childG: *parentG
```

Loading such a file would return a dictionary like:

```
{parentV: {parentG: {vx0, vx1, ...}},
 childV: {childV:{vx0, vx1, ...}}}
```

12.11.3 Versions with New File Formats

In the event that a future version of `SERPENT` causes substantial changes to the layout of output files, such that present readers will not work, then version-specific read methods will have to be written. If/when this happens, a procedure will be developed and added here.

MapStrVersions

This is a mapping variable located in the ResultsReader parser. The variable is a nested dictionary, where the keys describe the version and the nested dictionary describes the various data blocks within the results file (`_res.m`).

In general the results file is divided into three main blocks: - `metadata`: general description of the simulation - `resdata`: time-dependent values, e.g. `k-eff` - `universes`: universe dependent values, such as cross-sections The mapping through the variable `MapStrVersions` should reflect this.

The basic mapping definition relies on the following structure:

```
MapStrVersions = {'2.1.29':                                  # serpent version
{'meta': 'VERSION',                              # The starting keyword of the␣
↪metadata block
'rslt': 'MIN_MACROXS',           # The starting keyword of the resdata block
'univ': 'GC_UNIVERSE_NAME',      # The starting keyword of the universes block
'days': 'BURN_DAYS',                      # A keyword used in Serpent to describe time,␣
↪days
'burn': 'BURNUP',                # A keyword used in Serpent to describe burnup, MWd/
↪kgU
'infxs': 'INF_',                 # A prefix in Serpent used to describe infinite cross-
↪sections
'b1xs': 'B1_',                                   # A prefix in Serpent used to␣
↪describe b1 cross-sections
'varsUnc': ['MICRO_NG', 'MICRO_E', 'MACRO_NG', 'MACRO_E']} # Only the variables that␣
↪have no uncertainties in the universe block
```

In order to support different serpent versions, these keywords would need to be updated.

12.12 Comparison Methods

We are currently developing methods for our readers and containers to be able for comparing between like objects. This could be used to compare the effect of changing fuel enrichment on pin powers or criticality, or used to compare the effect of different SERPENT settings. The BaseObject that every object **should** inherit from contains the bulk of the input checking, so each reader and object needs to implement a private _compare method with the following structure:

```
def _compare(self, other, lower, upper, sigma):
    return <boolean output of comparison>
```

Note: While these methods will iterate over many quantities, and some quantities may fail early on in the test, the comparison method should continue until all quantities have been tested.

The value sigma should be used to compare quantities with uncertainties by constructing intervals bounded by $x \pm S\sigma$, where sigma$= S$. Quantities that do not have overlapping confidence windows will be considered too different and should result in a False value being returned from the method.

The lower and upper arguments should be used to compare values that do not have uncertainties. Both will be float values, with lower less than or equal to upper. This functionality is implemented with the serpentTools.utils.directCompare() function, while the result is reported with serpentTools.utils.logDirectCompare().

12.12.1 Use of messaging module

Below is a non-definitive nor comprehensive list of possible comparison cases and the corresponding message that should be printed. Using a range of message types allows the user to be able to easily focus on things that are really bad by using our *verbosity* setting.

- Two objects contain different data sets, e.g. different dictionary values - *warning()* displaying the missing items, and then apply test to items in both objects
- Two items are identically zero, or arrays of zeros - *debug()*
- Two items are outside of the sigma confidence intervals - *error()*
- Two items without uncertainties have relative difference
  - less than lower - *debug()*
  - greater than or equal to upper - *error()*
  - otherwise - *warning()*
- Two items are identical - *debug()*
- Two arrays are not of similar size - *error()*

12.12.2 High-level Logging and Comparison Utilities

The `utils` module contains a collection of functions that can be used to compare quantities and automatically log results. When possible, these routines should be favored over hand-writing comparison routines. If the situation calls for custom comparison functions, utilize or extend logging routines from *Low-level Logging Utilities* appropriately.

`serpentTools.utils.compare.`**`compareDictOfArrays`**(*d0*, *d1*, *desc*, *lower=0*, *upper=10*, *sigma=2*, *u0={}*, *u1={}*, *relative=True*)

High-level routine for evaluating the similarities of two dictionaries

The following tests are performed

1. Find a set of keys that both exist in `d0` and `d1` and point to arrays with identical shapes using *`getKeyMatchingShapes()`*
2. For each key in this common set, compare the values with *`logDirectCompare()`* or *`getLogOverlaps()`*. The latter is used if the key exists in `u0` and `u1`, provided uncertainty arrays are of identical shapes.

Parameters

- **d0** (*dict*) – Reference dictionary
- **d1** (*dict*) – Dictionarie to be compared to the reference
- **desc** (*str*) – Descption of the two dictionaries. Should describe what data they represent.
- **lower** (*float or int*) – Lower limit for relative tolerances in percent Differences below this will be considered allowable
- **upper** (*float or int*) – Upper limit for relative tolerances in percent. Differences above this will be considered failure and errors messages will be raised
- **sigma** (*int*) – Size of confidence interval to apply to quantities with uncertainties. Quantities that do not have overlapping confidence intervals will fail
- **u0** (*dict*) –
- **u1** (*dict*) – If uncKeys is not `None`, then find the uncertainties for data in `d0` and `d1` under the same keys.
- **relative** (*bool*) – If this evaluates to `true`, then uncertainties in `u0` and `u1` are relative.

Returns `True` If all comparisons pass

Return type bool

`serpentTools.utils.compare.`**`getCommonKeys`**(*d0*, *d1*, *quantity*, *desc0='first'*, *desc1='second'*, *herald=<function error>*)

Return a set of common keys from two dictionaries

Also supports printing warning messages for keys not found on one collection.

If `d0` and `d1` are `dict`, then the keys will be obtained with `d1.keys()`. Otherwise, assume we have an iterable of keys and convert to `set`.

Parameters

- **d0** (*dict or iterable*) –
- **d1** (*dict or iterable*) – Dictionary of keys or iterable of keys to be compared
- **quantity** (*str*) – Indicator as to what is being compared, e.g. `'metadata'`
- **desc0** (*dict or None*) –

- **desc1** (*dict or None*) – Description of the origin of each value set. Only needed if `quiet` evalues to `True`.
- **herald** (*callable*) – Function that accepts a single string argument used to notify that differences were found. If the function is not a callable object, a *serpentTools.messages.critical()* message will be printed and *serpentTools.messages.error()* will be used.

Returns Keys found in both `d{{0, 1}}`

Return type set

`serpentTools.utils.compare.`**directCompare**(*obj0*, *obj1*, *lower*, *upper*)

Return True if values are close enough to each other.

Wrapper around various comparision tests for strings, numeric, and arrays.

Parameters

- **obj0** (str or float or int or `numpy.ndarray`) –
- **obj1** (str or float or int or `numpy.ndarray`) – Objects to compare
- **lower** (*float or int*) – Lower limit for relative tolerances in percent Differences below this will be considered allowable
- **upper** (*float or int*) – Upper limit for relative tolerances in percent. Differences above this will be considered failure and errors messages will be raised
- **quantity** (*str*) – Description of the value being compared. Will be used to notify the user about any differences

Returns

Status code of the comparison.

- 0 - Values are identical to floating point precision or, for strings/booleans, are identical with the == operator
- 1 - Values are not identical, but the max difference is less than `lower`.
- 10 - Values differ, with the max difference greater than `lower` but less than `upper`
- 100 - Values differ by greater than or equal to `upper`
- 200 - Values should be identical (strings, booleans), but are not
- 250 - Numeric data has different shapes
- 255 - Values are of different types
- -1 - Type comparison is not supported. This means that developers should either implement a test for this data type, or use a different function

Return type int

See also:

- *logDirectCompare()* - Function that utilizes this and logs the results using the `serpentTools.messages` module

`serpentTools.utils.compare.`**logDirectCompare**(*obj0*, *obj1*, *lower*, *upper*, *quantity*)

Compare objects using *directCompare()* and log the result

Parameters

- **obj0** (str or float or int or numpy.ndarray) –
- **obj1** (str or float or int or numpy.ndarray) – Objects to compare
- **lower** (*float or int*) – Lower limit for relative tolerances in percent Differences below this will be considered allowable
- **upper** (*float or int*) – Upper limit for relative tolerances in percent. Differences above this will be considered failure and errors messages will be raised
- **quantity** (*str*) – Description of the value being compared. Will be used to notify the user about any differences

Returns `True` if the objects agree according to tolerances, or numerics differ less than `upper`. `False` otherwise

Return type bool

:raises TypeError:: If the objects being compared are not supported by *directCompare()*. Developers should either extend the function or utilize a different comparison function

See also:

- *directCompare()* - function that does the comparison
- *getOverlaps()* - function for evaluating values with uncertainties
- *getLogOverlaps()* - function that logs the result of stastistical comparisions

serpentTools.utils.compare.**splitDictByKeys**(*map0*, *map1*, *keySet=None*)

Return various sub-sets and dictionaries from two maps.

Used to test the internal workings on *getKeyMatchingShapes()*

Parameters

- **map0** (*dict*) –
- **map1** (*dict*) – Dictionaries to compare
- **keySet** (*set or None*) – Iterable collection of keys found in `map0` and `map1`. Missing keys will be returned from this function under the `missing0` and `missing1` sets. If `None`, take to be the set of keys that exist in both maps

Returns

- **missing0** (*set*) – Keys that exist in `keySet` but not in `map0`
- **missing1** (*set*) – Keys that exist in `keySet` but not in `map1`
- **differentTypes** (*dict*) – Dictionary with tuples `{key: (t0, t1)}` indicating the values `map0[key]` and `map1[key]` are of different types
- **badShapes** (*dict*) – Dictionary with tuples `{key: (t0, t1)}` indicating the values `map0[key]` and `map1[key]` are arrays of different shapes
- **goodKeys** (*set*) – Keys found in both `map0` and `map1` that are of the same type or point to arrays of the same shape

serpentTools.utils.compare.**getKeyMatchingShapes**(*map0*, *map1*, *quantity*, *keySet=None*, *desc0='first'*, *desc1='second'*)

Return a set of keys in map0/1 that point to arrays with identical shapes.

Parameters

- **keySet** (*set or list or tuple or iterable or None*) – Iterable container with keys that exist in map0 and map1. The contents of `map0/1` under these keys will be compared. If `None`, will be determined by *splitDictByKeys()*
- **map0** (*dict*) –
- **map1** (*dict*) – Two dictionaries containing at least all the keys in `keySet`. Objects under keys in `keySet` will have their sizes compared if they are `numpy.ndarray`. Non-arrays will be included only if their types are identical
- **quantity** (*str*) – Indicator as to what is being compared, e.g. `'metadata'`
- **desc0** (*str*) –
- **decs1** (*str*) – Descriptions of the two dictionaries being compared. Used to alert the user to the shortcomings of the two dictionaries

Returns Set of all keys that exist in both dictionaries and are either identical types, or are arrays of identical shapes

Return type set

See also:

- *splitDictByKeys()*

`serpentTools.utils.compare.`**getOverlaps**(*arr0*, *arr1*, *unc0*, *unc1*, *sigma*, *relative=True*)

Return the indicies of overlapping confidence intervals

Parameters

- **arr0** (`numpy.ndarray`) –
- **arr1** (`numpy.ndarray`) – Arrays containing the expected values to be compared
- **unc0** (`numpy.ndarray`) –
- **unc1** (`numpy.ndarray`) – Associated absolute uncertainties, 1σ, corresponding to the values in `arr0` and `arr1`
- **sigma** (*int*) – Size of confidence interval to apply to quantities with uncertainties. Quantities that do not have overlapping confidence intervals will fail
- **relative** (*bool*) – True if uncertainties are relative and should be multiplied by their respective values. Otherwise, assume values are absolute

Returns Boolean array of equal shape to incoming arrays. Every index with `True` as the value indicates that the confidence intervals for the arrays overlap at those indices.

Return type `numpy.ndarray`

Examples

Using absolute uncertainties:

```
>>> from numpy import ones, zeros, array
>>> a0 = ones(4)
>>> a1 = ones(4) * 0.5
>>> u0 = array([0, 0.2, 0.1, 0.2])
>>> u1 = array([1, 0.55, 0.25, 0.4])
```

Here, the first point in the confidence interval for `a0` is completely contained within that of `a1`. The upper limit of `a1[1]` is contained within the confidence interval for `a0`. The confidence intervals for the third point do not overlap, while the lower bound of `a0[3]` is within the confidence interval of `a1[3]`.

```
>>> getOverlaps(a0, a1, u0, u1, 1, relative=False)
array([True, True, False, True])
```

This function also works for multi-dimensional arrays as well.

```
>>> a2 = a0.reshape(2, 2)
>>> a3 = a1.reshape(2, 2)
>>> u2 = u0.reshape(2, 2)
>>> u3 = u1.reshape(2, 2)
>>> getOverlaps(a2, a3, u2, u3 1, relative=False)
array([[ True,  True],
       [False, False])
```

Raises **IndexError** – If the shapes of incoming arrays do not agree

See also:

- *getLogOverlaps()* - High-level function that uses this to report if two values have overlapping confidence intervals

serpentTools.utils.compare.**getLogOverlaps**(*quantity*, *arr0*, *arr1*, *unc0*, *unc1*, *sigma*, *relative=True*)

Wrapper around *getOverlaps()* that logs the result

Parameters

- **quantity** (*str*) – Name of the value being compared
- **arr0** (numpy.ndarray) –
- **arr1** (numpy.ndarray) –
- **unc0** (numpy.ndarray) –
- **unc1** (numpy.ndarray) – Arrays and their uncertainties to evaluate
- **sigma** (*int*) – Size of confidence interval to apply to quantities with uncertainties. Quantities that do not have overlapping confidence intervals will fail
- **relative** (*bool*) – If uncertainties are relative. Otherwise, assume absolute uncertainties.

Returns `True` if all locations `arr0` and `arr1` are either identical or within allowable statistical variations.

Return type bool

See also:

- *getOverlaps()* - This function performs all the comparisons while this function simply reports the output using `serpentTools.messages`

serpentTools.utils.docstrings.**compareDocDecorator**(*f*)

Decorator that updates doc strings for comparison methods.

Similar to `serpentTools.plot.magicPlotDocDecorator()` but for comparison functions

12.12.3 Low-level Logging Utilities

The `messages` module contains a collection of functions that can be used to notify the user about the results of a comparison routine.

`serpentTools.messages.`**`logIdentical`**(*obj0*, *obj1*, *quantity*)
Two objects are identical.

`serpentTools.messages.`**`logNotIdentical`**(*obj0*, *obj1*, *quantity*)
Values should be identical but aren't.

`serpentTools.messages.`**`logAcceptableLow`**(*obj0*, *obj1*, *quantity*)
Two values differ, but inside nominal and acceptable ranges.

`serpentTools.messages.`**`logAcceptableHigh`**(*obj0*, *obj1*, *quantity*)
Two values differ, enough to merit a warning but not an error.

`serpentTools.messages.`**`logOutsideTols`**(*obj0*, *obj1*, *quantity*)
Two values differ outside acceptable tolerances.

`serpentTools.messages.`**`logIdenticalWithUncs`**(*value*, *unc0*, *unc1*, *quantity*)
Notify that two values have identical expected values.

`serpentTools.messages.`**`logInsideConfInt`**(*value0*, *unc0*, *value1*, *unc1*, *quantity*)
Two values are within acceptable statistical limits.

`serpentTools.messages.`**`logOutsideConfInt`**(*value0*, *unc0*, *value1*, *unc1*, *quantity*)
Two values are outside acceptable statistical limits.

`serpentTools.messages.`**`logDifferentTypes`**(*type0*, *type1*, *quantity*)
Two values are of different types.

`serpentTools.messages.`**`logMissingKeys`**(*quantity*, *desc0*, *desc1*, *in0*, *in1*, *herald=<function error>*)
Log a warning message that two objects contain different items

Parameters

- **`quantity`** (*`str`*) – Indicator as to what is being compared, e.g. `'metadata'`
- **`desc0`** (*`str`*) –
- **`desc1`** (*`str`*) – Descriptions of the two originators
- **`in0`** (*`set or iterable`*) –
- **`in1`** (*`set or iterable`*) – Items that are unique to originators `0` and `1`, respectively
- **`herald`** (*`callable`*) – Callable function that accepts a single string. This will be called with the error message. If not given, defaults to *`error()`*

`serpentTools.messages.`**`logBadTypes`**(*quantity*, *desc0*, *desc1*, *types*)
Log an error message for containers with mismatched types

Parameters

- **`quantity`** (*`str`*) – Indicator as to what is being compared, e.g. `'metadata'`
- **`desc0`** (*`str`*) –
- **`desc1`** (*`str`*) – Descriptions of the two originators
- **`types`** (*`dict`*) – Dictionary where the keys represent the locations of items with mismatched types. Corresponding keys should be a list or tuple of the types for objects from `desc0` and `desc1` stored under `key`

serpentTools.messages.**logBadShapes**(*obj0*, *obj1*, *quantity*)

Log an error message that two arrays are of different shapes.

Parameters

- **obj0** (numpy.ndarray) –
- **obj1** (numpy.ndarray) – Arrays that have been compared and found to have different shapes
- **quantity** (*str*) – Descriptor of the quantity being compared, e.g. what these objects represent

CHAPTER

THIRTEEN

LICENSE

CHAPTER

FOURTEEN

DEVELOPER TEAM

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

- Andrew Johnson
- Dr. Dan Kotlyar
- Stefano Terlizzi
- Gavin Ridley
- Paul Romano

CHAPTER

FIFTEEN

GLOSSARY

Anaconda Python distrubution that makes installing packages like *numpy* easier for Windows. Includes over 700 packages. For a smaller, more minimal approach, see *Miniconda*. For more information, see https://www.anaconda.com/.

Anaconda prompt Modified Windows prompt to be used with the *Anaconda* python distrubution. Reconfigures python paths to point to packages installed via *conda*

conda From https://www.conda.io/docs/:

> Conda is an open source package management system and environment management system that runs on Windows, macOS, and Lunix. Conda quickly installs, runs, and updates packages and their dependncies

CI Continuous integration. Software we utilize to check that pull requests don't break the code.

git Distributed version control system. Allows us to test and implement new features with ease. For more information, see https://git-scm.com.

Jupyter notebook Powerful web application used in this project, for examples and tutorials containing real python code. From https://jupyter.org:

> The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text.

lint Bits of potentially erroneous code. Can be identified by a *linter*

linter Program that analyzes source code to check for errors, bugs, stylistic issues, and other potential hangups.

matplotlib Primary python package for plotting data. Highly customizable and extensible. More information at https://matplotlib.org

Miniconda Minimal installer for *conda*. From https://conda.io:

> Miniconda is a small, bootstrap version of Anaconda that only includes conda, Python, the packages they depend on, and a small number of useful packages

numpy Widely-used python package that allows multidimensional arrays and linear algebra routines. More information at https://www.numpy.org

pip Recommended tool for installing Python packages. More at https://pypi.org/project/pip/

pytest Fully featured python test runner. More at https://pytest.org/en/latest/

scipy Widely-used python package that contains more mathematical support and data structures, such as sparse matrices. Not required for this package, but allows the sparsity of some matrices to be exploited. More information at https://docs.scipy.org/doc/scipy/reference/

SERPENT Monte Carlo particle transport code developed at VTT Technical Research Centre of Finland, Ltd [serpent]. Produces output files that can be parsed by this project. More information, including distribution and licensing of `SERPENT` can be found at http://montecarlo.vtt.fi.

yaml Human-readable format used for configuration files in this project. For more information, see https://pyyaml.org/wiki/PyYAMLDocumentation

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[gpt] Aufiero, M. et al. “A collision history-based approach to sensitivity/perturbation calculations in the continuous energy Monte Carlo code SERPENT”, Ann. Nucl. Energy, 152 (2015) 245-258.

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

INDEX

Symbols

A

B

C

D

E

F

G

H

I

O

P

R

S

T

U

W

X

Y

Z