RESEARCH REPORT

VTT-R-00512-23

Group constant state point generation and fit goodness evaluation tools for KrakenTools

Authors: Vili-Arttu Ketola

Confidentiality: VTT Public

Version: 14.8.2023

Report's title	
Group constant state point generation and fit goodness evaluation tools for KrakenTools	
Customer, contact person, address	**Order reference**
SAFER 2028 (VYR)	SAFER 18/2023
Project name	**Project number/Short name**
DECAPOD 2023 / SAFER 2028	136053/DECAPOD
Author(s)	**Pages**
Vili-Arttu Ketola	18/–
Keywords	**Report identification code**
state point, group constant, fitting, polynomial, serpent, krakentools	VTT-R-00512-23

Summary

The group constant parametrization step of the Serpent-Ants calculation chain was previously lacking in polynomial model fit validation tools. Typically, either a narrow-range MOD5 model is used for all group constants, or some larger polynomial fit is defined arbitrarily. This can be a problem as it may be unclear whether the polynomial used is sufficiently accurate for representing the group constant values across state variables, burnups, and energy groups. This work involves the selection of Serpent state points to be used for polynomial fitting and validation, as well as the development of goodness-of-fit evaluation tools.

The state point generator "SPGenerator" class was implemented along with the "model_statistics.py" code. In this report, a summary of algorithms and functions are presented, as well as a simple demonstration case for a VVER440 reactor with some example results. Suggestions for how these results can be used for polynomial validation or new polynomial creation are also given.

Confidentiality	VTT Public

Espoo 14.8.2023

Written by	**Reviewed by**
Vili-Arttu Ketola, Research Trainee	Antti Rintala, Research Scientist

VTT's contact address

VTT Technical Research Centre of Finland Ltd, P.O. Box 1000, FI-02044 VTT, FINLAND

Distribution (customer and VTT)

SAFER2028 Technical Advisory Group 2.2
SAFER2028 Extranet

Approval

VTT TECHNICAL RESEARCH CENTRE OF FINLAND LTD

Date:	15 August 2023
Signature:	DocuSigned by: Silja Häkkinen FC3A155B9F4F479...
Name:	Silja Häkkinen
Title:	Research Team Leader

Contents

1. Introduction

The Serpent-Ants calculation chain includes several steps to complete neutronics simulations. Two of these steps were previously lacking in development: the generation or selection of state variable points for which to run the Serpent Monte Carlo code, and the validation of polynomial models used in the parametrization of group constant results. As the full group constant data can become very large even when condensed to a small number of energy groups, the parametrization step is vital to the calculation chain, and the model used should be as accurate as possible without over-fitting.

The 1999 paper by Peltonen has been a key reference for the consideration of a wide-range MOD6 polynomial model in place of the narrow-range MOD5.[1] Hence, an aim of this work is to reproduce a few aspects of Peltonen 1999 and to create tools to allow for the comparison of these models.

First, the implementation of a state point generator class and validation statistics code are presented—"SPGenerator" and "model_statistics.py" respectively—with summaries of algorithms and functions. Then, a simple demonstration case for a VVER440 reactor is outlined with a few example results, along with suggestions for how the results can be used for polynomial validation or new polynomial creation.

2. Implementation

2.1 State point generation

To begin, a suitable number of points for each relevant state variable must be selected. These are the momentary conditions, or branch cases, at which calculations are run in Serpent.[2] For the sake of simplicity, we only describe a case with one historical variation and no control rods, spacers, etc., as the implementation should be trivial to repeat for other histories. On the other hand, all valid combinations of state variables are included by the following algorithm in the SPGenerator class. See chapter 3 for a simple example case of generated state points for a VVER440 reactor model. The state point generator algorithms are designed to choose points that more-or-less replicate those chosen in Peltonen 1999.[1]

2.1.1 Coolant temperatures

First, the nominal and inlet coolant temperatures in Kelvin are required, and if the outlet temperature is not provided, it is estimated as

$$\mathrm{TCO}_{\mathrm{outlet}} = 2 \cdot \mathrm{TCO}_{\mathrm{nominal}} - \mathrm{TCO}_{\mathrm{inlet}} \tag{2.1}$$

Next, the room temperature point of 294 K is added, as well as a maximum at $\mathrm{TCO}_{\mathrm{outlet}} + 50\,\mathrm{K}$. Finally, all points in intervals of 50 K between 373 K and the inlet temperature are included. If the inlet temperature is lower than 373 K, then only the 373 K point is used. This provides a suitably wide range of coolant temperature points for use with any reasonably large polynomial.

The coolant temperature points are stored as a one-dimensional array datastructure.

2.1.2 Coolant densities

Coolant density points are different in that the nominal point is calculated by the algorithm instead of given. All density points are chosen according to the coolant temperatures (Kelvin) and pressures (Pascals). The nominal pressure, minimum coolant density, and number of density points to include for each coolant temperature are given as inputs, with the latter two optionally defaulting to $0.20\,\mathrm{g\,cm^{-3}}$ (as chosen in Peltonen 1999) and five points respectively.

For each coolant temperature, the coolant densities are a given number of equidistant points from the given minimum density to the saturation density, which is determined using the LibFluid package and the nominal pressure. However, the maximum temperature may result in a coolant which is a gas, in which case, the pressure is increased by one bar until the coolant is a liquid, and the coolant density is recalculated. Additionally, only one point is stored at the room temperature coolant—the density at atmospheric pressure. An example set of coolant points can be seen in Figure 1.

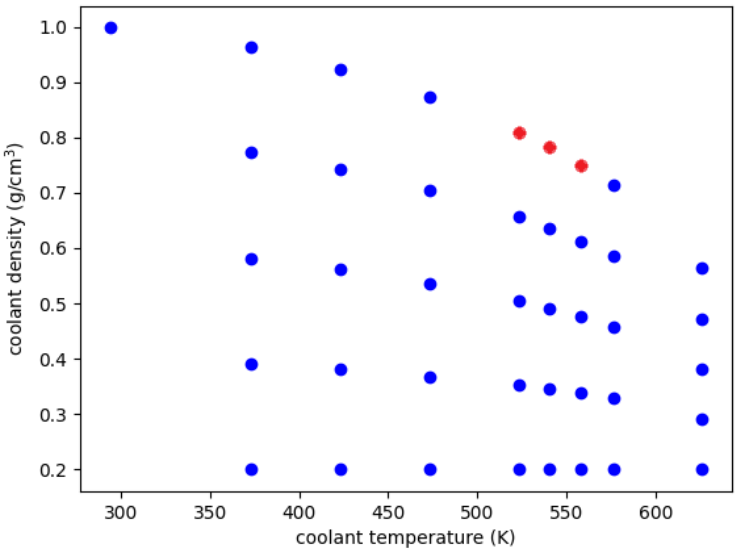


Figure 1. Coolant temperature and density state points as presented by SPGenerator for the VVER440 demonstration. Each temperature has the default of five density points between the density at saturation pressure and the default minimum density. A single point at room temperature and atmospheric pressure is included. An example of typical narrow-range points used for MOD5 polynomial fitting are marked in red.

The coolant densities expand the array of coolant temperature points to two dimensions, with every valid combination of coolant points such that the number of points is the product of coolant temperature and density points. Invalid points in this datastructure and further ones are stored as NaN values.

2.1.3 Fuel temperatures

Nominal and maximum fuel temperatures in Kelvin are used as input for picking fuel temperature points in addition to the previous state points array. First, one particular point at $TFU_{nominal}$ + 300 K is added if it is not over the given maximum. Then, every coolant temperature point is also added to the list of fuel temperatures as long as the fuel temperature is greater than or equal to the coolant temperature. An example can be seen in Figure 2.

Fuel temperatures extend the datastructure again to three dimensions to get every combination of coolant temperatures, densities, and fuel temperatures.

2.1.4 Boron concentrations

Nominal and maximum boron concentrations in parts per million are used as input. The algorithm simply adds a zero boron point, as well as twice the nominal if it is under the given maximum.

As before, the array is extended to four dimensions. After boron points are added, the algorithm is complete and the array is flattened to just a single list of Serpent Branch cards which can be conveniently used with GCGenerator to create Serpent inputs.

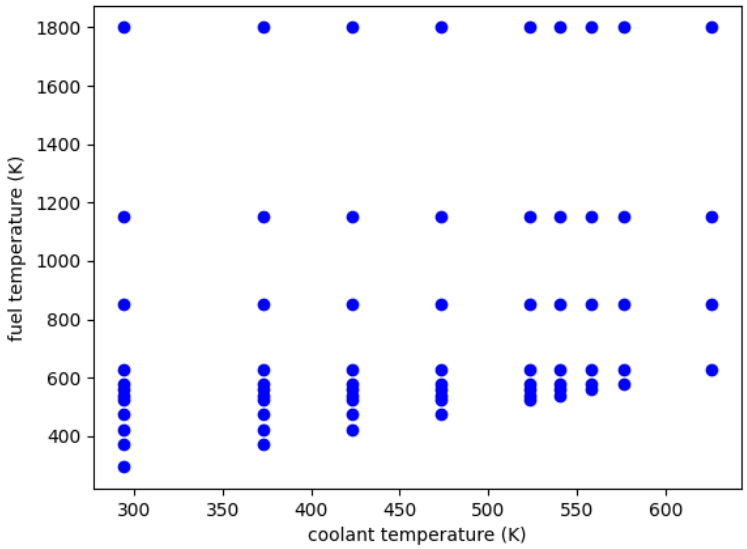


Figure 2. Coolant and fuel temperature points. All coolant temperature points are included as fuel temperatures under the condition that the fuel temperature is greater than or equal to the coolant temperature. Additional fuel temperature points are added.

2.1.5 Validation points

SPGenerator also has the feature of creating additional points to be used for polynomial model validation and statistical tests. A validation set is necessary so that the points used for fitting the polynomials are not the same as the points used to evaluate their goodness-of-fit.

The generate_between_points method processes the list of cases and uses a helper function to simply take the midpoint between all data set points. Furthermore, for each state variable, one additional point is added near the middle of this new validation set such that the deviation is relatively small—so that it can be used for estimating the slope of the polynomial during validation:

$$x_{\text{new}} = x_{\text{mid}} + \frac{x_{\text{mid}} - x_{\text{mid+1}}}{50} \tag{2.2}$$

This factor of $1/50$ is chosen arbitrarily, and further testing should be used to determine how close the point should be to be as accurate as possible. Given the stochastic nature of Serpent, there is a limit on how small the deviation can be as the significance of random variation increases.

This validation set is returned as another list of Serpent cases which can be run separately. Polynomials are fit only to the full set of state points, whereas validation is conducted with this smaller set.

2.2 Polynomial fitting

As discussed in chapter 1, it is important to fit a polynomial to the group constant data to condense it for further calculations. The independent variables are state variable terms with x-data consisting of the state points from SPGenerator, and a single polynomial is used to model all group constants separately across burnups, different energy group condensations, momentary “select” variables, etc. Note that in the demonstration of this work, the full set of state points is calculated in Serpent, and narrow-range models use a filtered set of the full points for the fit.

The codes introduced in this report do not change how polynomials are fit; however, it is suggested to fit three or more different polynomials for validation. Alternatively, one fit can be used, and then validation can be run repeatedly while making improvements to it with each step.

Referring to the red points displayed in Figure 1, polynomial models can be described as either narrow-range or wide-range fits. In the case of narrow-range fits such as MOD5, a set number of state points is used for the parametrization, unlike wide-range fits which may use all available state points. A full example of narrow-range points can be found in Table 1 of the 2023 Valtavirta and Rintala paper.[3]

Models may have some peculiarities which need to be treated differently in the codes. Namely, there are two differences: First, MOD5 uses BORDENS2 for cross-terms of boron density, which is the same boron state variable input as the typical BORDENS, but the nominal point is instead defined as zero. Second, if the fit is intended to represent only liquid water at a constant pressure, the coolant temperature TCO is not an independent variable. Hence, for the demonstration case in chapter 3, TCO is not included in MOD5, because the coolant density points at maximum coolant temperature which use a higher pressure (described in section 2.1.2) are not included in the narrow-range set of points.

2.3 Model validation

The model_statistics.py file is the larger set of codes presented, containing statistical calculations and various helper functions for the reading and processing of raw and model data.

2.3.1 Helper functions

There are several new short helper functions and algorithms for processing the data.

2.3.1.1 get_gc_name

This is a very short function that, for a given genpoly.py group constant, returns the name of the group constant as displayed in the XS files from a dictionary. There has previously been no single dictionary of all group constant names. Table 3 displays this dictionary.

2.3.1.2 read_coefs

After a model has been fit to the Serpent output data and written to an XS file, this function is used to read all the coefficients of the polynomial for a particular group constant, burnup, and energy group. This is needed especially when working with data matrices for statistical tests.

2.3.1.3 read_powers

Similar to 2.3.1.2 read_coefs, this function parses some of the XS file output in order to read the “fb_table” which contains the degrees of exponents of each term of the polynomial.

2.3.1.4 create_data_matrix

This function creates a statistical data matrix X for the given points and powers from 2.3.1.3 read_powers. It does this by reading each state variable x-value from 2.3.1.5 get_x_values, and matching the correct independent variable values to their exponent. The data matrix is useful in simplifying multivariate statistical calculations.

Each column of X represents a different term in the polynomial, beginning with a column full of ones for the constant term if included. Each row, then, represents a data point. A small data matrix example representing the polynomial and all combinations of the data below is shown:

$$f(\boldsymbol{x}) = a_0 + a_1 x_0 + a_2 x_0^2 + a_3 x_1 + a_4 x_0 x_1 \tag{2.3}$$

$$x_0 = 2, 4 \qquad\qquad x_1 = 3, 9$$

$$\mathrm{X}_f = \begin{bmatrix} 1 & 2 & 4 & 3 & 6 \\ 1 & 4 & 16 & 3 & 12 \\ 1 & 2 & 4 & 9 & 18 \\ 1 & 4 & 16 & 9 & 36 \end{bmatrix}$$

2.3.1.5 get_x_values

Given the state variable x-data, this function extracts and returns the desired Δx value (which is then used in the polynomials) with the help of 2.3.1.6 read_scale_factors. These Deltas are typically calculated as:

$$\Delta x = (x - \text{A1}) \cdot \text{A2} \tag{2.4}$$

where A1 (nominal point) and A2 are the scale factors from 2.3.1.6.

2.3.1.6 read_scale_factors

This function extracts the A1 and A2 scale factors by parsing the given XS file for the “fb_scale” table. The scale factors for all possible state variables are returned in index orders.

2.3.1.7 get_raw_data

This function reads all the raw Serpent fuel data for all given group constants at a particular given burnup point. It does this by extracting data from the Serpent output files with the read_all_gc_results method, and then condenses the data to two energy groups if requested and filters the node branch list. This filtered node branch list is returned, as well as a new list datastructure containing just group constant points and point state variable data.

This process typically takes the longest amount of time in the calculation with larger data sets, so this method is designed to only be called once for each burnup and for the results to then be filtered later for desired points in validation.

2.3.1.8 get_model_data

For a given range of state variable points, this function reads the XS file to evaluate the polynomial for a particular group constant. This is done using the genpoly.py evaluate_at method. Compared to 2.3.1.7 get_raw_data, this method is quick to call, and so it is done for each group constant individually for simplicity rather than attempting to filter a larger datastructure afterwards.

2.3.2 Validation functions

With these helper functions, some model validation statistics can be calculated and other helpful results such as plots can be created. It is important to note that many validation functions assume that the user wishes to validate only fuel temperature, boron, and coolant density. This is because, in the demonstrations, coolant temperature and other state variables are not used in MOD5, which is focal for comparisons. This will be discussed in chapter 3, along with suggestions for how these validation function results can be used.

2.3.2.1 get_rms_and_rss

Given a validation raw data set and model data at matching state variable points, this function calculates normalized root-mean-square deviations (RMS or NRMSD) and the residual sum of squares (RSS).

The RSS value, as the name implies, is simply the sum of squared differences between the model fit f_i and raw data points y_i:

$$\text{RSS} = \sum^{n}(y_i - f_i) \tag{2.5}$$

The RSS is important in later calculations, such as 2.3.2.2 get_vifs, and it is also used for the RMS.

$$\text{RMSD} = \sqrt{\frac{\text{RSS}}{n}} = \sqrt{\frac{\sum^{n}(y_i - f_i)}{n}} \tag{2.6}$$

Normalization is done by the mean of model data:

$$\text{NRMSD} = \frac{\text{RMSD}}{\bar{f}} \tag{2.7}$$

2.3.2.2 get_vifs

This function calculates the variance inflation factor (VIF) for every term in the polynomial. The VIF is essentially a fitting of one independent variable in a linear model against all other independent variables. Thus, it is a quantity describing the multicollinearity of the term in question.

The VIF calculation algorithm loops over all terms in the given polynomial. Using the data matrix X, the algorithm runs an ordinary least square regression for each independent variable against all the others with scipy.linalg.lstsq.[4] Then, the coefficient of determination is calculated from the residual and total sum of squares:

$$r^2 = 1 - \frac{\sum(y_i - f_i)^2}{\sum(y_i - \overline{y})^2} \tag{2.8}$$

Finally, the VIF of the term is simply

$$\mathrm{VIF} = \frac{1}{1 - r^2} \tag{2.9}$$

2.3.2.3 term_diagnostics

This function returns the p-values for two-tailed t-tests for each term in the polynomial a_i, for which the null and alternate hypotheses respectively are

$$H_0 : a_i = 0 \qquad H_0 : a_i \neq 0 \tag{2.10}$$

In other words, this function calculates the statistical significance of terms in the polynomial.

The calculation of t-scores starts with the M-matrix:

$$\begin{aligned} \mathrm{M} &= \left(\mathrm{X}^{\mathrm{T}}\mathrm{X}\right)^{-1} \\ \boldsymbol{m} &= \mathrm{diag}(\mathrm{M}) \end{aligned} \tag{2.11}$$

Next, the standard deviations are also needed: The variation can be calculated from the RSS values in 2.3.2.1 get_rms_and_rss, so that

$$s^2 = \frac{\mathrm{RSS}}{n - k} \tag{2.12}$$

where n is the number of validation points and k is the number of terms in the polynomial. Then, the standard deviations for each term are, conveniently,

$$\boldsymbol{s} = \sqrt{s^2} \cdot \sqrt{\boldsymbol{m}} \tag{2.13}$$

from which we can calculate all the t-scores:

$$\boldsymbol{t} = \frac{\boldsymbol{b}}{\boldsymbol{s}} \tag{2.14}$$

where $\boldsymbol{b}$ contains the coefficients given by 2.3.1.2 read_coefs. Finally, p-values are obtained from scipy.stats.t.[5]

2.3.2.4 validate_derivatives

This function calculates the derivative of the given polynomial analytically as well as the slope between the two nearest points in the validation data. This is done for three scenarios: Varying fuel temperature with constant coolant and boron density at their respective nominal points, varying coolant density with constant fuel temperature and boron density at nominal points, and varying boron density with fuel temperature and coolant density at nominal points. The derivatives and slopes are returned together for comparison.

2.3.2.5 gc_plotter

This function formats and generates plots for three scenarios where one state variable is varied and the others are kept constant, just like in 2.3.2.4 validate_derivates. The plots show given raw data as a scatter plot, and model data as a curve. This function is called for a specific burnup, and generates plots where all energy groups are included, and also for each energy group separately, allowing for a closer look at how the model is fitting to the data.

2.3.2.6 evaluate_k

Given a node branch list from Serpent and an instance of the parametrization class from genpoly.py, this function evaluates the multiplication factor four times: First, k_∞ is obtained from the node branch list. Second, the infinite k-eigenvalue is calculated according to the parametrization using the genpoly.py evaluate_k_at method. Third and fourth, the node branch k is calculated with all poisons included and removed using the groupconstants.utils.py evaluate_k_from_node_branch function. This evaluation of multiplication factors is repeated for all points and returned as four lists.

2.3.2.7 Information criteria

While not defined as a function in the model_statistics.py code, it is recommended to calculate the Akaike and Bayesian information criteria (AIC and BIC respectively) for the results. This can be done from the RSS values from 2.3.2.1 get_rms_and_rss:

$$\mathrm{AIC} = 2k + n\log\left(\frac{\mathrm{RSS}}{n-k}\right) \qquad \mathrm{BIC} = n\log\left(\frac{\mathrm{RSS}}{n}\right) + k\log(n) \tag{2.15}$$

These AIC and BIC values are slightly differently calculated quantities used for expressing the relative “goodness-of-fit” of the model while punishing for over-fitting. This allows for a direct comparison of multiple model fits. Lower values indicate better models.

3. Demonstration

3.1 Serpent input

The VVER440 case is used as a simple example input, with no control rods, spacers, or other geometric variations in Serpent. This input can be found in the *groupconstant_tools* Git-repository.[6] Serpent is run for 73 burnup points between and including points 0.0 and 60.0, however, validation will be completed with only three: 0.0, 30.0, and 60.0.

The VVER440-specific values required as inputs for SPGenerator are listed below.

- nominal coolant pressure: $1.23 \cdot 10^7$ Pa
- inlet coolant temperature: 540 K
- nominal coolant temperature: 558 K
- outlet coolant temperature: 576 K
- nominal fuel temperature: 850 K
- maximum fuel temperature: 1800 K
- nominal boron concentration: 500 ppm
- maximum boron concentration: 2275 ppm

Additionally, the nominal coolant density is obtain from the libfluid package, which is roughly 0.7506 g cm^{-3}. With this information, the generate_points method in the SPGenerator class can be called to obtain all the cases needed to create Serpent inputs with GCGenerator.

For these inputs, a full list of 1248 state points cases is created. Some of these state points can be seen in Figures 1 and 2. The code can then be run again for validation points using the SPGenerator generate_between_points method instead. In this case, we have 420 validation points in addition to the 1248 state points. For simplicity, only one history is used: the nominal one. It would be trivial to run all the codes again for more histories.

3.2 Parametrization

For the VVER440 example, three polynomial models are used for parametrization. The first fit is the MOD5-based model containing the terms shown in Table 1. It may be important to notice the lack of a coolant temperature term, the use of TFUSQRT for fuel temperature, and the use of BORDENS2 in addition to BORDENS for boron densities. As mentioned previously, the MOD5 fit also uses a narrow-range set of points rather than the full 1248 state points points.

The second fit is based on MOD6; however, it contains only 29 terms since only those relating to fuel temperature, coolant temperature and density, and boron density are used. That is, the bypass coolant density, xenon density, and plutonium history terms are not included for the sake of simplicity. The terms are shown in Table 2, adapted from Peltonen 1999.[1]

The third fit is an intentionally poor, over-fit polynomial used as a reference. The name of this model is presented as ‘ALL’. It contains all possible combinations of the state variables up to the fourth order, for a total of 68 terms.

3.3 Results

3.3.1 Running the validation

The validation is run for the 21 group constants displayed in Table 3, though some of them require special treatment by the calculation codes. It is also run for the three aforementioned burnup points, as well as both the full eight energy groups and a condensation to two energy groups. Only results from the two-group case are shown, since they are easier to interpret physically.

To run the validation codes, the basic datastructures can be extracted from the previously listed model_statistics.py functions. As discussed, some validation functions require filtered lists which only contain points that vary for one state variable at a time, whereas others require the full points. Appendix A contains an example of running some of the validation methods for fuel temperatures only. The example produces some of the statistics, which could be processed further or output to files as necessary. Loops can also be utilized to run this over all desired burnups, energy group condensations, group constants, etc. with ease.

Table 1. The terms of the MOD5-based group constant polynomial model, where a_i represents coefficients.

Term	Description
a_0	constant nominal point
$a_1 \cdot$ TFUSQRT	first order fuel temperature
$a_2 \cdot$ TFUSQRT^2	second order fuel temperature
$a_3 \cdot$ DCO	first order coolant density
$a_4 \cdot$ DCO^2	second order coolant density
$a_5 \cdot$ BORDENS	first order boron density
$a_6 \cdot$ BORDENS^2	second order boron density
$a_7 \cdot$ DCO $\cdot$ BORDENS2	boron density and coolant density cross-term

3.3.2 Interpretation of results

The functions presented were chosen to create a selection of validation results which can be used to identify problems with polynomial fits or even improve a single fit. Some group constant validation results are presented here for a burnup of 30.0 and condensed to two energy groups. The full set of codes, plots, and result outputs is located in the results folder of the *groupconstant_tools* Git-repository.[6]

3.3.2.1 NRMSD and plots

For validating a single model fit, the NRMSD values and plots can be used to spot any obvious problems in the fit. Consider the diffusion coefficient displayed in Figure 3. The diffusion coefficient is expected to be most dependent on the coolant density only, and so the plots should match very well for that state variable, and not so well for the other two—for both the MOD5 and MOD6 models.

The slope comparisons from 2.3.2.4 validate_derivatives can also provide a first look into any issues with the model. Consider the example below, for the diffusion coefficient at 30.0 burnup and energy group two:

```
analytical derivatives vs point derivatives for TFU, DCO, and BOR respectively:
 MOD5:
  ([0.3948, -0.3830, 0.0127], [-3.0782e-06, -0.6739, 1.8501e-06])
  absolute differences: [0.3948 0.2908 0.0127]
 MOD6:
  ([-0.005147, -0.3370, 2.2947e-05], [-3.0782e-06, -0.6739, 1.8501e-06])
  absolute differences: [5.1444e-03 0.3682 2.1097e-05]
```

It can be seen that the absolute differences in derivatives is generally much greater for MOD5, suggesting that there may be an issue with that fit. However, the derivative validation codes are still limited as mentioned in the future work section 4.2 and also in 2.1.5.

When comparing multiple model fits, the NRMSD values and plots can be used to get a basic visual idea of which is better. In the case of Figure 3, it seems that both the MOD5 and MOD6 polynomials represent the group constant well. Consider also the following 2.3.2.1 get_rms_and_rss output values which correspond to the figure for MOD5 and MOD6 respectively:

```
gc DIFF, nrmsd for energy group 1:
 '0.0003185',    '0.0001585'
```

Here it can be seen that both fits are very strong, though the normalized RMS deviation of MOD6 is roughly half that of MOD5. It could be concluded that MOD5 is most likely sufficient for representing the diffusion coefficient specifically, though interpretations may very based on user necessity.

3.3.2.2 Information criteria

The AIC and BIC can provide some deeper insight into the relative goodness-of-fit values, especially since they punish for over-fitting. Consider the following example values (rounded to integers) for the MOD5, MOD6, and ALL fits respectively, nothing that lower values are better.

```
gc DIFF MOD5:    MOD6:            ALL:
 AIC = -4026      AIC = -6695      AIC = -5485
 BIC = -4002      BIC = -6608      BIC = -4864
```

Here it is clear that, when comparing the models for the diffusion coefficient, the MOD6 model is potentially stronger than both the MOD5 and ALL fits, suggesting that the MOD5 polynomial is most likely under-fit, and the ALL polynomial is over-fit.

The information criteria can also be useful when creating a new polynomial. When making changes such as adding or removing terms, the AIC and BIC provide a good indication for whether or not the model has been improved.

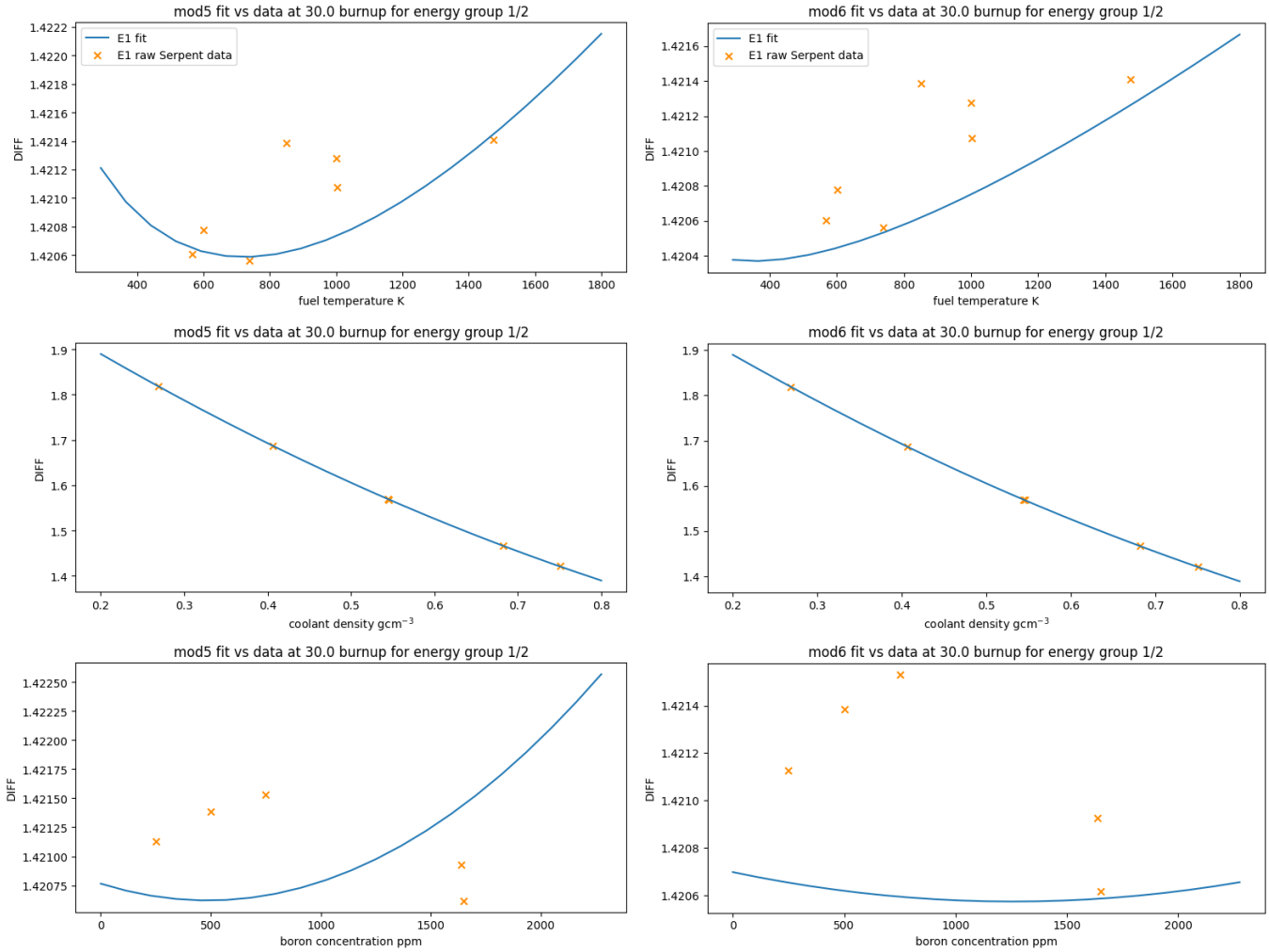


Figure 3. Diffusion coefficient results from 2.3.2.5 gc_plotter for a single burnup point and energy group. Left: MOD5 polynomial fit. Right: MOD6 polynomial fit.

3.3.2.3 Term diagnostics and VIF

The term diagnostic p-values can be used in conjunction with variance inflation factors to get an idea for which terms may be unnecessary to the model. Consider the following p-values for the diffusion coefficient for the MOD5 polynomial at 30.0 burnup:

```
gc DIFF
 [0.0000 0.7750 0.7100 2.2741e-172 1.1457e-61 0.9969 0.8705 0.9713]
```

Here, the terms correspond in order to those in Table 1. Regardless of which reasonable cutoff point is chosen, it can be seen that the null hypothesis can be rejected only for the constant nominal point term and two coolant density terms. In other words, they are extremely significant in representing the diffusion coefficient. When viewing Figure 3 as well, we can see that these term diagnostic results make sense. It should be noted that, since the same polynomial is used for all group constants, a term should be insignificant over all group constants before it can safely be dropped.

Each model also has one VIF value for every term. The VIF is used to check for multicollinearity. If a specific term appears to be unnecessary for the model according to term diagnostics, the VIF can be checked to see if that term is already well-represented by the others, in which case it could be dropped. For MOD5:

```
fit mod5
 [0, 1.4767, 1.4767, 29.5111, 15.3946, 23.8382, 12.9874, 10.0905]
```

A VIF of 5 is typically used as a cutoff point, and 10 is already considered extremely high. In this example, it can be seen that: although the fuel temperature terms were insignificant for modeling the diffusion coefficient according to the p-values, they are not very correlated with the other terms in the model and thus may still be useful to keep in the polynomial. It is also important to note that a high VIF score alone does not necessarily justify dropping a term from the model.

In summary, one could start from a higher-order polynomial, and using the term diagnostics and VIF values, determine which polynomial terms are never significant across all group constants and burnups. Then, after eliminating those terms and running the validation again, one can check the AIC and BIC values for improvement. Alternatively, the NRMSD values and plots can be used to see if some lower-order or narrow-range polynomial is good enough for some limited purposes of a user.

4. Conclusion

4.1 Summary

In this work, codes were developed to allow for an easier selection of state points to use as Serpent inputs, and to evaluate any polynomial fits to the resultant group constant data. It is important to use a variety of statistics to gain a holistic evaluation of the model, and the kinds of results generated were chosen so that it may be easy to both validate existing models and develop better ones if necessary. Using these tools, the best polynomial can be determined statistically, though it should be expected to vary for different input cases and necessities of the user.

Codes were written with standard practices including comments, self-documentation, etc. However, there are still improvements that can be made if these codes should be included in KrakenTools.

4.2 Future work

Potential future work includes improvements to the functions already presented here as well as additional suggestions. Example results from the VVER440 demonstration suggest that the MOD6 fit is the best one between the three that were tested; however, further testing is required for different inputs and also to see how these models could be further developed.

One major improvement is the automation of validation codes such that term diagnostics and VIFs would be referenced across all group constants and burnups, so that all insignificant polynomial terms can be identified at once. Additionally, a code allowing for quickly repeated validation following minor changes to the polynomials could be useful.

Referring back to Figure 2, it can be seen that some state points are close enough together that they are most likely not necessary. A new function could be written to automatically identify and remove such points.

The validation process should also be repeated with different inputs and models, in order to verify that calculations are correct and work as expected. Runs should be completed with multiple histories, fuel types, and select momentary variations such as control rods and spacers. Certain algorithms may require adjustments too, such as the nearest points in the validation set used for 2.3.2.4 validate_derivatives. The distance between nearest points should be appropriate so that they are close enough to represent the tangent of the polynomial, but not so close that they are significantly affected by the stochastic nature of the data.

References

[1] Jyrki Peltonen. *Laaja-alainen vaikutusalamalli*. IVO Power Engineering, 1999. Report YDIN-GT3-41.

[2] Group constant generation for square lattice SMR tutorial. *Kraken Wiki*, 2023. Available at https://serpent.vtt.fi/kraken/index.php/Group_constant_generation_for_square_lattice_SMR_tutorial. Accessed 11 August 2023.

[3] Ville Valtavirta and Antti Rintala. Validating Kraken for VVER-1000 fuel cycle simulations using the X2 benchmark. *Annals of Nuclear Energy*, 190:109878, 2023.

[4] scipy.linalg.lstsq. *SciPy documentation*, 2023. Available at https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lstsq.html. Accessed 11 August 2023.

[5] scipy.stats.t. *SciPy documentation*, 2023. Available at https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.t.html. Accessed 11 August 2023.

[6] Vili-Arttu Ketola. groupconstant_tools. 2023. Available at https://ringworld.vtt.fi/scm/git/groupconstant_tools. Accessed 11 August 2023.

A. Raw code

An example for running some of the validation functions. Results could then be stored or processed further.

```
import numpy as np

from model_statistics import get_raw_data, get_model_data
from model_statistics import get_rms_and_rss, term_diagnostics, get_vifs
from krakentools.groupconstants.genpoly.definitions import GroupConstants


# --- Choose which assembly to get the group constant for
identifier = 'VVER440'
# --- Choose the target group constant
gc = GroupConstants.DIFF
# --- Values for "select" variables control rod, grid, instrumentation
select_vars = ['0','0','0']
# --- Choose the burnup to get the data for
burnup = 30.0
# --- Get results condensed to 2 energy groups
E_groups = 2
# --- Define the file name extension to use for the fit
fit = 'mod5'
# --- Specify the location of the XS file from fitting the fuel
xs_file = './fuel/{}_{}g_{}.xs'.format(identifier,E_groups,fit)

# --- Get the validation data points (in this case 13 points)
raw_val_points, nbl = get_raw_data(gc, 'val_fuel/VVER440', burnup, condense=True)

# --- Filter the data for this gc specifically
raw_val_data = []
for raw_val_point in raw_val_points:
    if raw_val_point[2] != gc:
        continue
    raw_val_data.append(raw_val_point)

# --- Next steps: Collect the parametrization model data too. We get it for
#     each independent variable of interest, and for both the validation and
#     full set of points

# --- Record the range of points used in the validation data set, so we can get
#     model points at the same values for statistical tests
tfu_val_range = []
dco_val_range = []
bor_val_range = []
```

```
# --- Check each point and add the state variable point if not already included
for raw_val_point in raw_val_points:
    if raw_val_point[0][0] not in tfu_val_range:
        tfu_val_range.append(raw_val_point[0][0])

    if raw_val_point[0][3] not in dco_val_range:
        dco_val_range.append(raw_val_point[0][3])

    if raw_val_point[0][1] not in bor_val_range:
        bor_val_range.append(raw_val_point[0][1])

# --- Get model data for the validation set for those point ranges
model_val_data = get_model_data(identifier, gc, burnup, select_vars, xs_file,
                                tfu_val_range, dco_val_range, bor_val_range,
                                bordens2=True)

# --- Start evaluating model statistics

# --- Collect normalized RMS-deviation and RSS values
nrmsds, rss_values = get_rms_and_rss(raw_val_data, model_val_data,
                                     E_groups, use_tco_flag=False)

# --- Number of terms and number of validation data points
k = 8
n = 420

# --- Get the combined RSS
rss = np.sum(rss_values)

# --- Calculate, for example, Akaike and Bayesian information criteria
aic = 2*k + n*np.log(rss / (n-k))
bic = n*np.log(rss/n) + k*np.log(n)

# --- Get term diagnostic p-values
p = term_diagnostics(raw_val_data, rss, gc, burnup, xs_file, k)

# --- Get and output VIF value here. They are specific to the fit and
#     independent of other factors.

# --- Calculate Variance inflation factor (VIF) for every term
vifs = get_vifs(raw_val_data, xs_file)
```

Table 2. The terms of the MOD6-based group constant polynomial model, where a_i represents coefficients, modified from Peltonen 1999 to match this VVER440 demonstration case.

Term	Description
a_0	constant nominal point
$a_1 \cdot \mathrm{DCO} + a_2 \cdot \mathrm{DCO}^2 + a_3 \cdot \mathrm{DCO}^3$	coolant density
$a_4 \cdot \mathrm{TCO} + a_5 \cdot \mathrm{TCO}^2$	coolant temperature
$a_6 \cdot \mathrm{TFUSQRT} + a_7 \cdot \mathrm{TFUSQRT}^2$	fuel temperature
$a_8 \cdot \mathrm{BORDENS} + a_9 \cdot \mathrm{BORDENS}^2$	boron density
$a_{10} \cdot \mathrm{DCO} \cdot \mathrm{TCO} + a_{11} \cdot \mathrm{DCO} \cdot \mathrm{TFUSQRT} +$ $a_{12} \cdot \mathrm{DCO} \cdot \mathrm{BORDENS}$	second-order coolant density cross-terms
$a_{13} \cdot \mathrm{DCO} \cdot \mathrm{TCO} \cdot \mathrm{BORDENS} + a_{14} \cdot \mathrm{DCO}^2 \cdot \mathrm{TCO} +$ $a_{15} \cdot \mathrm{DCO}^2 \cdot \mathrm{TFUSQRT} + a_{16} \cdot \mathrm{DCO}^2 \cdot \mathrm{BORDENS}$	third-order coolant density cross-terms
$a_{17} \cdot \mathrm{TCO} \cdot \mathrm{TFUSQRT} + a_{18} \cdot \mathrm{TCO} \cdot \mathrm{BORDENS}$	coolant temperature cross-terms
$a_{19} \cdot \mathrm{DCO}^4 + a_{20} \cdot \mathrm{DCO}^5$	higher-order coolant density
$a_{21} \cdot \mathrm{DCO} \cdot \mathrm{TCO}^2 + a_{22} \cdot \mathrm{DCO} \cdot \mathrm{TCO} \cdot \mathrm{TFUSQRT} +$ $a_{23} \cdot \mathrm{DCO} \cdot \mathrm{TCO} \cdot \mathrm{TFUSQRT} + a_{24} \cdot \mathrm{DCO} \cdot \mathrm{TFUSQRT}^2 +$ $a_{25} \cdot \mathrm{DCO}^2 \cdot \mathrm{TCO}^2$	higher-order coolant density cross-terms
$a_{26} \cdot \mathrm{TCO}^2 \cdot \mathrm{BORDENS} + a_{27} \cdot \mathrm{TCO} \cdot \mathrm{TFUSQRT}^2 +$ $a_{28} \cdot \mathrm{TCO} \cdot \mathrm{BORDENS}^2 + a_{29} \mathrm{TFUSQRT} \cdot \mathrm{BORDENS}$	other higher-order cross-terms

Table 3. Group constants included in the validation, displayed with both genpoly.py and Ants file names.

genpoly.py GC name	Ants XS file GC name
SIGA	Sigma_a
NUSF	nuSigma_f
KSF	kappaSigma_f
SIGF	Sigma_f
DIFF	D
INVV	1/v
CHI	chi
SIDE_DF	side_df
BETA	beta
LAMBDA	lambda
G_I135	gamma_I135
G_XE135	gamma_Xe135
G_XE135M	gamma_Xe135m
G_PM149	gamma_Pm149
G_SM149	gamma_Sm149
SIG_MICRO_ABS_I135	sigma_I135_a
SIG_MICRO_ABS_XE135	sigma_Xe135_a
SIG_MICRO_ABS_XE135M	sigma_Xe135m_a
SIG_MICRO_ABS_PM149	sigma_Pm149_a
SIG_MICRO_ABS_SM149	sigma_Sm149_a

DocuSign

Certificate Of Completion

Envelope Id: CC0D1E970F764BF88646C6948D7F16CB
Status: Completed
Subject: Complete with DocuSign: VTT-R-00512-23.pdf
Source Envelope:
Document Pages: 19
Signatures: 1
Certificate Pages: 1
Initials: 0
AutoNav: Enabled
EnvelopeId Stamping: Enabled
Time Zone: (UTC+02:00) Helsinki, Kyiv, Riga, Sofia, Tallinn, Vilnius

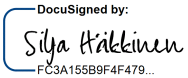
Envelope Originator:
Christina Vähävaara
Vuorimiehentie 3, Espoo,
., . P.O Box1000,FI-02044
Christina.Vahavaara@vtt.fi
IP Address: 130.188.40.124

Record Tracking

Status: Original
15 August 2023 | 07:15

Holder: Christina Vähävaara
Christina.Vahavaara@vtt.fi

Location: DocuSign

Signer Events	Signature	Timestamp
Silja Häkkinen silja.hakkinen@vtt.fi Research Team Leader Security Level: Email, Account Authentication (None), Authentication	DocuSigned by:  FC3A155B9F4F479... Signature Adoption: Pre-selected Style Using IP Address: 130.188.17.16	Sent: 15 August 2023 | 07:17 Viewed: 15 August 2023 | 09:05 Signed: 15 August 2023 | 09:06

Authentication Details
SMS Auth:
Transaction: 71d869ea-2394-499b-852d-9097d31715a3
Result: passed
Vendor ID: TeleSign
Type: SMSAuth
Performed: 15 August 2023 | 09:05
Phone: +358 40 0236898

Electronic Record and Signature Disclosure:
Not Offered via DocuSign

In Person Signer Events	Signature	Timestamp

Editor Delivery Events	Status	Timestamp

Agent Delivery Events	Status	Timestamp

Intermediary Delivery Events	Status	Timestamp

Certified Delivery Events	Status	Timestamp

Carbon Copy Events	Status	Timestamp

Witness Events	Signature	Timestamp

Notary Events	Signature	Timestamp

Envelope Summary Events	Status	Timestamps
Envelope Sent	Hashed/Encrypted	15 August 2023 | 07:17
Certified Delivered	Security Checked	15 August 2023 | 09:05
Signing Complete	Security Checked	15 August 2023 | 09:06
Completed	Security Checked	15 August 2023 | 09:06

Payment Events	Status	Timestamps