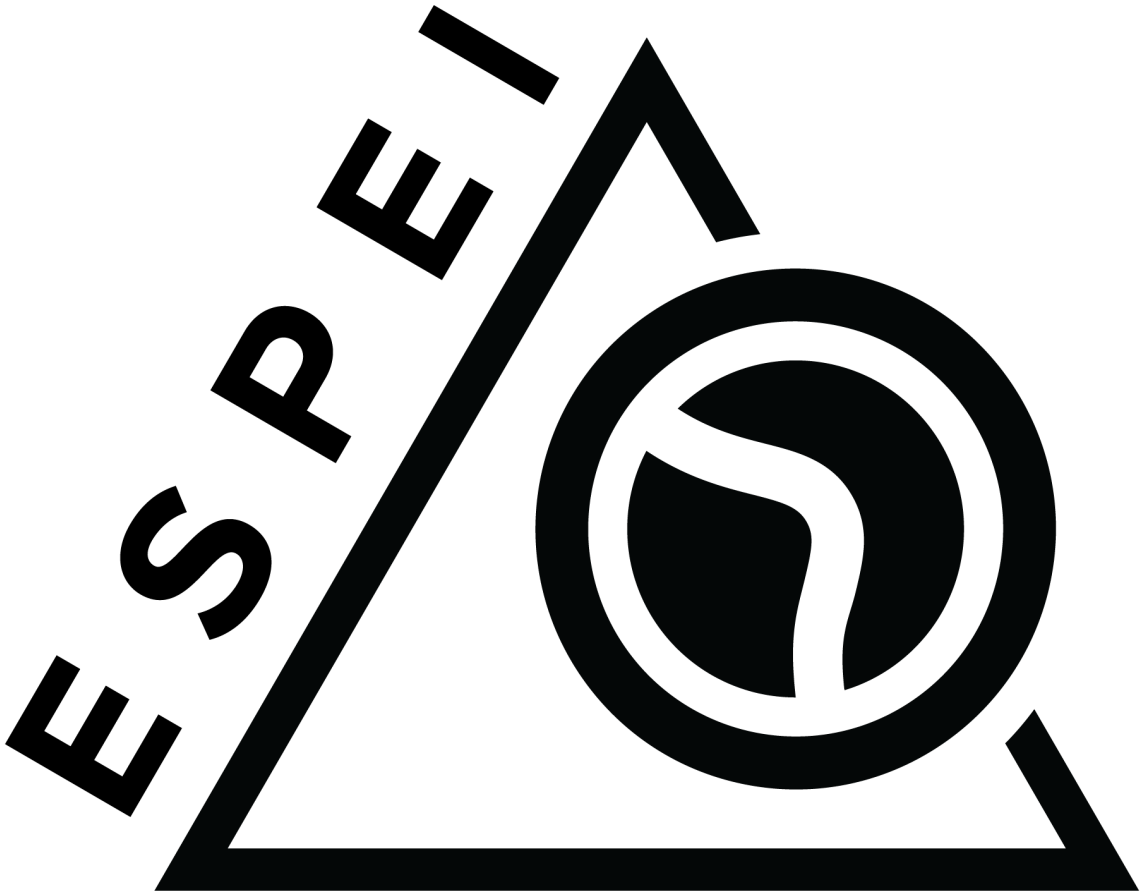


ESPEI Documentation

Release 0.7.4

Brandon Bocklund

Dec 18, 2019

Contents

Part I

Introduction

CHAPTER 1

Installation

1.1 Anaconda

Installing ESPEI from PyPI (by `pip install espei`) is **not** supported. Please install ESPEI using Anaconda.

```
conda install -c pycalphad -c conda-forge --yes espei
```

After installation, you must turn off dask's work stealing. Change the work stealing setting to `distributed.scheduler.work-stealing: False` in dask's configuration. See *configuration* below for more details.

1.2 Development versions

To make changes to the ESPEI source code, the development version must be installed. If you'll need to make changes to pycalphad simultaneously, follow the instructions to install the development version of pycalphad first.

To install the latest development version of ESPEI, use Anaconda to download ESPEI and all of the required dependencies, then remove the installed release version of ESPEI and replace it with the latest version from GitHub:

```
git clone https://github.com/phasesresearchlab/espei.git
cd espei
conda install -c pycalphad -c conda-forge espei
conda remove --force espei
pip install --no-deps -e .
```

Upgrading ESPEI later requires you to run `git pull` in this directory.

After installation, you must turn off dask's work stealing. Change the work stealing setting to `distributed.scheduler.work-stealing: False` in dask's configuration. See *configuration* below for more details.

1.3 Configuration

ESPEI uses dask-distributed to parallelize ESPEI.

After installation, you must turn off dask's work stealing! Change the your dask configuration file to look something like:

```
distributed:
  version: 2
  scheduler:
    work-stealing: False
```

The configuration file paths can be found by running `from espei.utils import get_dask_config_paths; get_dask_config_paths()` in a Python interpreter. If multiple configurations are found, the latter configurations take precendence over the former, so check them from last to first. See the dask-distributed documentation for more.

CHAPTER 2

Quickstart

ESPEI has two different fitting modes: single-phase and multi-phase fitting. You can run either of these modes or both of them sequentially.

To run either of the modes, you need to have a phase models file that describes the phases in the system using the standard CALPHAD approach within the compound energy formalism. You also need to describe the data that ESPEI should fit to. You will need single-phase and multi-phase data for a full run. Fit settings and all datasets are stored as JSON files and described in detail at the *Making ESPEI datasets* page. All of your input datasets should be validated by running `espei --check-datasets my-input-datasets`, where `my-input-datasets` is a folder of all your JSON files.

The main output result is going to be a database (defaults to `out.tdb`), an array of the steps in the MCMC trace (defaults to `trace.npy`), and the an array of the log-probabilities for each iteration and chain (defaults to `lnprob.npy`).

2.1 Single-phase only

If you have only heat capacity, entropy and enthalpy data and mixing data (e.g. from first-principles), you may want to see the starting point for your MCMC calculation.

Create an input file called `espei-in.yaml`.

```
system:
  phase_models: my-phases.json
  datasets: my-input-datasets
generate_parameters:
  excess_model: linear
  ref_state: SGTE91
```

Then ESPEI can be run by running

```
espei --input espei-in.yaml
```

2.2 Multi-phase only

If you have a database already and just want to do a multi-phase fitting, you can specify a starting TDB file (named `my-tdb.tdb`) with

```
system:
  phase_models: my-phases.json
  datasets: my-input-data
mcmc:
  iterations: 1000
  input_db: my-tdb.tdb
```

The TDB file you input must have all of the degrees of freedom you want as FUNCTIONs with names beginning with `VV`.

2.3 Restart from previous run-phase only

If you've run an MCMC fitting already in ESPEI and have a trace file called `my-previous-trace.npy` , then you can resume the calculation with the following input file

```
system:
  phase_models: my-phases.json
  datasets: my-input-data
mcmc:
  iterations: 1000
  input_db: my-tdb.tdb
  restart_trace: my-previous-trace.npy
```

2.4 Full run

A minimal full run of ESPEI with single phase fitting and MCMC fitting is done by the following

```
system:
  phase_models: my-phases.json
  datasets: my-input-data
generate_parameters:
  excess_model: linear
  ref_state: SGTE91
mcmc:
  iterations: 1000
```

2.5 Input Customization

ESPEI lets you control many aspects of your calculations with the input files shown above. See *ESPEI YAML input files* for a full description of all possible inputs.

2.6 FAQ

2.6.1 Q: There is an error in my JSON files

A: Common mistakes are using single quotes instead of the double quotes required by JSON files. Another common source of errors is misaligned open/closing brackets.

Many mistakes are found with ESPEI's `check-datasets` utility. Run `espei check-datasets my-input-datasets` on your directory `my-input-datasets`.

2.6.2 Q: How do I analyze my results?

A: By default, ESPEI will create `trace.npy` and `lnprob.npy` for the MCMC chain at the specified save interval and according to the save interval (defaults to ever iteration). These are created from arrays via `numpy.save()` and can thus be loaded with `numpy.load()`. Note that the arrays are preallocated with zeros. These filenames and settings can be changed using in the input file. You can then use these chains and corresponding log-probabilities to make corner plots, calculate autocorrelations, find optimal parameters for databases, etc.. Finally, you can use py:mod:*espei.plot* functions such as `multiplot` to plot phase diagrams with your input equilibria data and `plot_parameters` to compare single-phase data (e.g. formation and mixing data) with the properties calculated with your database.

2.6.3 Q: Can I run ESPEI on a supercomputer supporting MPI?

A: Yes! ESPEI has MPI support. See the *Advanced Schedulers* page for more details.

2.6.4 Q: How is the log probability reported by ESPEI calculated?

MCMC simulation requires determining the probability of the data given a set of parameters, $p(D|\theta)$. In MCMC, the log probability is often used to avoid floating point errors that arise from multiplying many small floating point numbers. For each type of data the *error*, often interpreted as the absolute difference between the expected and calculated value, is determined. For the types of data and how the error is calculated, refer to the ESPEI paper[1].

The error is assumed to be normally distributed around the experimental data point that the prediction of a set of parameters is being compared against. The log probability of each data type is calculated by the log probability density function of the error in this normal distribution with a mean of zero and the standard deviation as given by the data type and the adjustable weights (see `data_weights` in *ESPEI YAML input files*). The total log probability is the sum of all log probabilities.

Note that any probability density function always returns a positive value between 0 and 1, so the log probability density function should return negative numbers and the log probability reported by ESPEI should be negative.

ESPEI YAML input files

2.6.5 Q: Why is the version of ESPEI '0+unknown'?

A: A version number of `'0+unknown'` indicates that you do not have git installed. This can occur on Windows where git is not in the PATH (and the Python interpreter cannot see it). You can install git using `conda install git` on Windows.

[1] B. Bocklund, R. Otis, A. Egorov, A. Obaied, I. Roslyakova, Z.-K. Liu, ESPEI for efficient thermodynamic database development, modification, and uncertainty quantification: application to Cu-Mg, (2019). http://arxiv.org/abs/1902.01269.

CHAPTER 3

References

ESPEI, or Extensible Self-optimizing Phase Equilibria Infrastructure, is a tool for automated thermodynamic database development within the CALPHAD method. It uses pycalphad for calculating Gibbs free energies of thermodynamic models.

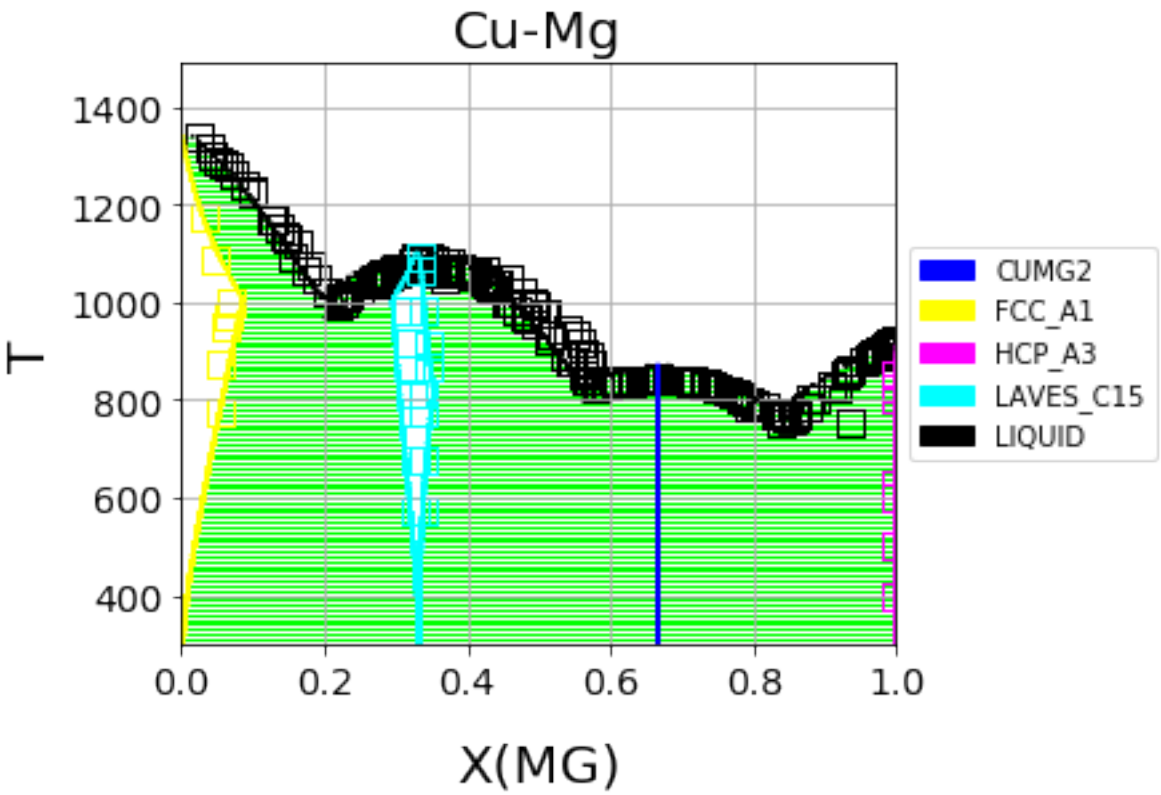


Fig. 1: Cu-Mg phase diagram from a database created with and optimized by ESPEI. See the *Cu-Mg Example*.

CHAPTER 4

Goals

1. Offer a free and open-source tool for users to develop multicomponent databases with quantified uncertainty
2. Enable development of CALPHAD-type models for Gibbs energy, thermodynamic or kinetic properties
3. Provide a platform to build and apply novel model selection and optimization methods

The implementation for ESPEI involves first fitting single-phase data by calculating parameters in thermodynamic models that are linearly described by the single-phase input data. Then Markov Chain Monte Carlo (MCMC) is used to optimize the candidate models from the single-phase fitting to multi-phase zero-phase fraction data. The benefit of this approach is the automated, simultaneous fitting for many parameters that yields uncertainty quantification, as shown in Otis and Liu High-Throughput Thermodynamic Modeling and Uncertainty Quantification for ICME. Jom 69, (2017). Single-phase and multi-phase fitting methods are described in Chapter 3 of Richard Otis's thesis.

CHAPTER 5

History

The ESPEI package is based on a fork of pycalphad-fitting. The name and idea of ESPEI are originally based off of Shang, Wang, and Liu, ESPEI: Extensible, Self-optimizing Phase Equilibrium Infrastructure for Magnesium Alloys Magnes. Technol. 2010 617-622 (2010).

CHAPTER 6

Change log

6.1 What's New

6.1.1 0.7.4 (2019-12-09)

This release includes small fixes for parameter generation.

- Excluded model contributions are fixed for models with different sublattice site ratios and for data that are not endmembers (@bocklund - #113)

6.1.2 0.7.3 (2019-12-02)

This change includes several new features and performance improvements.

- Drop Python 2 support (Python 2 is no longer supported on January 1, 2020).
- Update dask and distributed support to versions >=2. (@bocklund)
- Users can tweak the AICc penalty factor for each phase to nudge parameter selection towards adding more or fewer parameters based on user modeling intitution. (@bocklund)
- Allow for tracefile and probfile to be set to None. (@jwsiegel2510)
- Weighting individual datasets in single phase fitting is now implemented via scikit-learn. (@bocklund)
- Performance improvements by reducing overhead. (@bocklund)
- Increased solver accuracy by using pycalphad's exact Hessian solver. (@bocklund)
- Support writing SER reference state information to the *ELEMENT* keyword in TDBs based on the SGTE unary 5 database. (@bocklund)
- MCMC now calculates the likelihood of the initial parameter set so the starting point can be reasonably compared. (@bocklund)
- Fixed a bug where mis-aligned configurations and site occupancies in single phase datasets passed the dataset checker (@bocklund)

6.1.3 0.7.2 (2019-06-12)

This is a small bugfix release that fixes the inability to provide the EmceeOptimizer a `restart_trace`.

6.1.4 0.7.1 (2019-06-03)

This is a significant update reflecting many internal improvements, new features, and bugfixes. Users using the YAML input or the `run_espei` Python API should see entirely backwards compatible changes with ESPEI 0.6.2.

pycalphad 0.8, which introduced many key features for these changes is now required. This should almost completely eliminate the time to build phases due to the symengine backend (phases will likely build in less time than to call the MCMC objective function). Users can expect a slight performance improvement for MCMC fitting.

Improvements

- Priors can now be specified and are documented online.
- Weights for different datasets are added and are supported by a `"weight"` key at the top level of any dataset.
- Weights for different types of data are added. These are controlled via the input YAML and are documented there.
- A new internal API is introduced for generic fitting of parameters to datasets in the `OptimizerBase` class. The MCMC optimizer in emcee was migrated to this API (the `mcmc_fit` function is now deprecated, but still works until the next major version of ESPEI). A simple SciPy-based optimizer was implemented using this API.
- Parameter selection can now be passed initial databases with parameters (e.g. for adding magnetic or other parameters manually).
- pycalphad's reference state support can now be used to properly reference out different types of model contributions (ideal mixing, magnetic, etc.). This is especially useful for DFT thermochemical data which does not include model contributions from ideal mixing or magnetic heat capacity. Useful for experimental data which does include ideal mixing (previously ESPEI assumed all data
- Datasets and input YAML files now have a tag system where tags that are specified in the input YAML can override any keys/values in the JSON datasets at runtime. This is useful for tagging data with different weights/model contribution exclusions (e.g. DFT tags may get lower weights and can be set to exclude model contributions). If no tags are applied, removing ideal mixing from all thermochemical data is applied automatically for backwards compatibility. This backwards compatibility feature will be removed in the next major version of ESPEI (all model contributions will be included by default and exclusions must be specified manually).

Bug fixes

- Bug fixed where asymmetric ternary parameters were not properly replaced in SymPy
- Fixed error where ZPF error was considering the chemical potentials of stoichiometric phases in the target hyperplane (they are meaningless)
- Report the actual file paths when dask's work-stealing is set to false.
- Errors in the ZPF error function are no longer swallowed with -np.inf error. Any errors should be reported as bugs.
- Fix bug where subsets of symbols to fit are not built properly for thermochemical data

Other

- Documentation recipe added for *plot_parameters*
- [Developer] ZPF and thermochemical datasets now have an function to get all the data up front in a dictionary that can be used in the functions for separation of concerns and calculation efficiency by not recalculating the same thing every iteration.
- [Developer] a function to generate the a context dict to pass to lnprob now exists. It gets the datasets automatically using the above.
- [Developer] transition to pycalphad's new build_callables function, taking care of the `v.N` state variable.
- [Developer] Load all YAML safely, silencing warnings.

6.1.5 0.6.2 (2018-11-27)

This backwards-compatible release includes several bug fixes and improvements.

- Updated branding to include the new ESPEI logo. See the logo in the `docs/_static` directory.
- Add support for fitting excess heat capacity.
- Bug fix for broken potassium unary.
- Documentation improvements for recipes
- pycalphad 0.7.1 fixes for dask, sympy, and gmpy2 should mean that ESPEI should not require package upgrade or downgrades. Please report any installations issues in *ESPEI's Gitter Channel <https://gitter.im/PhasesResearchLab/ESPEI>*.
- [Developers] ESPEI's `eq_callables_dict` is now `pycalphad.codegen.callables.build_callables`.
- [Developers] matplotlib plotting tests are removed because nose is no longer supported.

6.1.6 0.6.1 (2018-08-28)

This a major release with several important features and bug fixes.

- Enable use of ridge regression alpha for parameter selection via the `parameter_generation.ridge_alpha` input parameter.
- Add ternary parameter selection. Works by default, just add data.
- Set memory limit to zero to avoid dask killing workers near the dask memory limits.
- Remove ideal mixing from plotting models so that `plot_parameters` gives the correct entropy values.
- Add recipes documentation that contains some Python code for common utility operations.
- Add documentation for running custom distributed schedulers in ESPEI

6.1.7 0.6 (2018-07-02)

This is a update including *breaking changes to the input files* and several minor improvements.

- Update input file schema and Python API to be more consistent so that the `trace` always refers to the collection of chains and `chain` refers to individual chains. Additionally removed some redundancy in the parameters nested under the `mcmc` heading, e.g. `mcmc_steps` is now `iterations` and `mcmc_save_interval` is now `save_interval` in the input file and Python API. See Writing Input documentation for all of the updates.
- The default save interval is now 1, which is more reasonable for most MCMC systems with significant numbers of phase equilibria.
- Bug fixes for plotting and some better plotting defaults for plotting input data
- Dataset parsing and cleaning improvements.
- Documentation improvements (see the PDF!)

6.1.8 0.5.2 (2018-04-28)

This is a major bugfix release for MCMC multi-phase fitting runs for single phase data.

- Fixed a major issue where single phase thermochemical data was always compared to Gibbs energy, giving incorrect errors in MCMC runs.
- Single phase errors in ESPEI incorrectly compared values with ideal mixing contributions to data, which is excess only.
- Fixed a bug where single phase thermochemical data with that are dependent on composition and pressure and/or temperature were not fit correctly.
- Added utilities for analyzing ESPEI results and add them to the Cu-Mg example docs.

6.1.9 0.5.1 (2018-04-17)

This is a minor bugfix release.

- Parameter generation for phases with vacancies would produce incorrect parameters because the vacancy site fractions were not being correctly removed from the contributions due to their treatment as `Species` objects in `pycalphad >=0.7`.

6.1.10 0.5 (2018-04-03)

- Parameter selection now uses the corrected AIC, which further prevents overparameterization where there is sparse training data.
- Activity and single phase thermochemical data can now be included in MCMC fitting runs. Including single phase data can help anchor metastable phases to DFT data when they are not on the stable phase diagram. See the Gathering input data documentation for information on how to input activity data.
- Dataset checking has been improved. Now there are checks to make sure sublattice interactions are properly sorted and mole fractions sum to less than 1.0 in ZPF data.
- Support for fitting phases with arbitrary pycalphad Models in MCMC, including (charged and neutral) species and ionic liquids. There are several consequences of this:
  - ESPEI requires support on `pycalphad >=0.7`
  - ESPEI now uses pycalphad `Model` objects directly. Using the JIT compiled Models has shown up to a *50% performance improvement* in MCMC runs.

  - Using JIT compiled `Model` objects required the use of `cloudpickle` everywhere. Due to challenges in overriding `pickle` for upstream packages, we now rely solely on `dask` for scheduler tasks, including `mpi` via `dask-mpi`. Note that users must turn off `work-stealing` in their `~/.dask/config.yaml` file.

- [Developers] Each method for calculating error in MCMC has been moved into a module for that method in an `error_functions` subpackage. One top level function from each module should be imported into the `mcmc.py` and used in `lnprob`. Developers should then just customize `lnprob`.
- [Developers] Significant internal docs improvements: all non-trivial functions have complete docstrings.

6.1.11 0.4.1 (2018-02-05)

- Enable plotting of isothermal sections with data using `dataplot` (and `multiplot`, etc.)
- Tielines are now plotted in `dataplot` for isothermal sections and T-x phase diagrams
- Add a useful `ravel_conditions` method to unpack conditions from datasets

6.1.12 0.4 (2017-12-29)

- MCMC is now deterministic by default (can be toggled off with the `mcmc.deterministic` setting).
- Added support for having no scheduler (running with no parallelism) with the `mcmc.scheduler` option set to `None`. This may be useful for debugging.
- Logging improvements
  - Extraneous warnings that may be confusing for users and dirty the log are silenced.
  - A warning is added for when there are no datasets found.
  - Fixed a bug where logging was silenced with the dask scheduler
- Add `optimal_parameters` utility function as a helper to get optimal parameter sets for analysis
- Several improvements to plotting
  - Users can now plot phase diagram data alone with `dataplot`, useful for checking datasets visually. This changes the API for `dataplot` to no longer infer the conditions from an equilibrium `Dataset` (from pycalphad). That functionality is preserved in `eqdataplot`.
  - Experimental data points are now plotted with unique symbols depending on the reference key in the dataset. This is for both phase diagram and single phase parameter plots.
  - Options to control plotting parameters (e.g. symbol size) and take user supplied Axes and Figures in the plotting functions. The symbol size is now smaller by default.
- Documentation improvements for API and separation of theory from the Cu-Mg example
- Fixes a bug where elements with single character names would not find the correct reference state (which are typically named GHSERCC for the example of C).
- [Developer] All MCMC code is moved from the `paramselect` module to the `mcmc` module to separate these tasks
- [Developer] Support for arbitrary user reference states (so long as the reference state is in the `refdata` module and follows the same format as SGTE91)

6.1.13 0.3.1.post2 (2017-10-31)

- Propagate the new entry point to setup.py

6.1.14 0.3.1.post1 (2017-10-31)

- Fix for module name/function conflict in entry point

6.1.15 0.3.1 (2017-10-31)

- ESPEI is much easier to run interactively in Python and in Jupyter Notebooks
- Reference data is now included in ESPEI instead of in pycalphad
- Several reference data fixes including support for single character elements ('V', 'B', 'C', ...)
- Support for using multiprocessing to parallelize MCMC runs, used by default (@olivia-higgins)
- Improved documentation for installing and developing ESPEI

6.1.16 0.3.post2 (2017-09-20)

- Add input-schema.yaml file to installer

6.1.17 0.3.post1 (2017-09-20)

- Add LICENSE to manifest

6.1.18 0.3 (2017-09-20)

- **ESPEI input is now described by a file.** This change is breaking. Old command line arguments are not supported. See Writing input files for a full description of all the inputs.
- New input options are supported, including modifying the number of chains and standard deviation from the mean
- ESPEI is now available on conda-forge
- TinyDB 2 support is dropped in favor of TinyDB 3 for conda-forge deployment
- Allow for restarting previous mcmc calculations with a trace file
- Add Cu-Mg example to documentation

6.1.19 0.2.1 (2017-08-17)

Fixes to the 0.2 release plotting interface

- `multiplot` is renamed from `multi_plot`, as in docs.
- Fixed an issue where phases in datasets, but not in equilibrium were not plotted by dataplot and raised an error.

6.1.20 0.2 (2017-08-15)

- New `multiplot` interface for convenient plotting of phase diagrams + data. `dataplot` function underlies key data plotting features and can be used with `eqplot`. See their API docs for examples. Will break existing code using multiplot.
- MPI support for local/HPC runs. Only single node runs are explicitly supported currently. Use `--scheduler='MPIPool'` command line option. Requires `mpi4py`.
- Default debug reporting of acceptance ratios
- Option (and default) to output the log probability array matching the trace. Use `--probfile` option to control.
- Optimal parameters are now chosen based on lowest error in chain.
- Bug fixes including
  - py2/3 compatibility
  - Unicode datasets
  - handling of singular matrix errors from pycalphad's `equilibrium`
  - reporting of failed conditions

6.1.21 0.1.5 (2017-08-02)

- Significant error checking of JSON inputs.
- Add new `--check-datasets` option to check the datasets at path. It should be run before you run ESPEI fittings. All errors must be resolved before you run.
- Move the espei script module from `fit.py` to `run_espei.py`.
- Better docs building with mocking
- Google docstrings are now NumPy docstrings

6.1.22 0.1.4 (2017-07-24)

- Documentation improvements for usage and API docs
- Fail fast on JSON errors

6.1.23 0.1.3 (2017-06-23)

- Fix bad version pinning in setup.py
- Explicitly support Python 2.7

6.1.24 0.1.2 (2017-06-23)

- Fix dask incompatibility due to new API usage

6.1.25 0.1.1 (2017-06-23)

- Fix a bug that caused logging to raise if bokeh isn't installed

6.1.26 0.1 (2017-06-23)

ESPEI is now a package! New features include

- Fork https://github.com/richardotis/pycalphad-fitting
- Use emcee for MCMC fitting rather than pymc
- Support single-phase only fitting
- More control options for running ESPEI from the command line
- Better support for incremental saving of the chain
- Control over output with logging over printing
- Significant code cleanup
- Better usage documentation

See what's new!

Part II

Tutorials

CHAPTER 7

Cu-Mg example

7.1 Cu-Mg Example

The Cu-Mg binary system is an interesting and simple binary subsystem for light metal alloys. It has been modeled in the literature by Coughanowr et al.[1], Zuo and Chang[2] and Zhou et al.[3] and was featured as a case study in Computational Thermodynamics The Calphad Method by Lukas, Fries, & Sundman[4].

Here we will combine density functional theory and experimental calculations of single-phase data to generate a first-principles phase diagram. Then that database will be used as a starting point for a Markov Chain Monte Carlo (MCMC) Bayesian optimization of the parameters to fit zero-phase fraction data.

7.1.1 Input data

All of the input data for ESPEI is stored in a public ESPEI-datasets repository on GitHub. The data in this repository is Creative Commons Attribution 4.0 (CC-BY-4.0) licensed and may be used, commercialized or reused freely.

In order to run ESPEI with the data in ESPEI-datasets, you should clone this repository to your computer. Files referred to throughout this tutorial are found in the *CU-MG* folder. The input files will be very breifly explained in this tutorial so that you are able to know their use. A more detailed description of the files is found on the *Making ESPEI datasets* page.

If you make changes or additions, you are encouraged to share these back to the ESPEI-datasets repository so that others may benefit from this data as you have. You may then add your name to the CONTRIBUTORS file as described in the README.

[1] Coughanowr, C. A., Ansara, I., Luoma, R., Hamalainen, M. & Lukas, H. L. Assessment of the Cu-Mg system. Zeitschrift f{ü}r Met. 82, 574–581 (1991).

[2] Zuo, Y. U. E. & Chang, Y. A. Thermodynamic calculation of the Mg-Cu phase diagram. Zeitschrift f{ü}r Met. 84, 662–667 (1993).

[3] Zhou, S. et al. Modeling of Thermodynamic Properties and Phase Equilibria for the Cu-Mg Binary System. J. Phase Equilibria Diffus. 28, 158–166 (2007). doi:10.1007/s11669-007-9022-0

[4] Lukas, H., Fries, S. G. & Sundman, B. Computational Thermodynamics The Calphad Method. (Cambridge University Press, 2007). doi:10.1017/CBO9780511804137

7.1.2 Phases and CALPHAD models

The Cu-Mg system contains five stable phases: Liquid, disordered fcc and hcp, the C15 Laves phase and the CuMg2 phase. All of these phases will be modeled as solution phases, except for CuMg2, which will be represented as a stoichiometric compound. The phase names and corresponding sublattice models are as follows:

```
LIQUID:     (CU, MG)1
FCC_A1:     (CU, MG)1  (VA)1
HCP_A3:     (CU, MG)1  (VA)1
LAVES_C15:  (CU, MG)2  (CU, MG)1
CUMG2:      (CU)1  (MG)2
```

These phase names and sublattice models are described in the JSON file *Cu-Mg-input.json* file as seen below

```
{
  "components": ["CU", "MG", "VA"],
  "phases": {
         "LIQUID" : {
            "sublattice_model": [["CU", "MG"]],
            "sublattice_site_ratios": [1]
         },
         "CUMG2": {
            "sublattice_model": [["CU"], ["MG"]],
            "sublattice_site_ratios": [1, 2]
         },
         "FCC_A1": {
            "sublattice_model": [["CU", "MG"], ["VA"]],
            "sublattice_site_ratios": [1, 1]
         },
         "HCP_A3": {
            "sublattice_site_ratios": [1, 0.5],
            "sublattice_model": [["CU", "MG"], ["VA"]]
         },
         "LAVES_C15": {
            "sublattice_site_ratios": [2, 1],
            "sublattice_model": [["CU", "MG"], ["CU", "MG"]]
         }
    }
}
```

7.1.3 ESPEI

ESPEI has two types of fitting – parameter generation and MCMC optimization. The parameter generation step uses experimental and first-principles data of the derivatives of the Gibbs free energy to parameterize the Gibbs energies of each individual phase. The MCMC optimization step fits the generated parameters to experimental phase equilibria data. These two fitting procedures can be used together to fully assess a given system. For clarity, we will preform these steps separately to fit Cu-Mg. The next two sections are devoted to describing ESPEI's parameter generation and optimization.

Though it should be no problem for this case, since the data has already been used, you should get in the habit of checking datasets before running ESPEI. ESPEI has a tool to help find and report problems in your datasets. This is automatically run when you load the datasets, but will fail on the first error. Running the following commmand (assuming from here on that you are in the `CU-MG` folder from ESPEI-datasets):

```
espei --check-datasets input-data
```

The benefit of the this approach is that all of the datasets will be checked and reported at once. If there are any failures, a list of them will be reported with the two main types of errors being `JSONError`, for which you should read the JSON section of *Making ESPEI datasets*, or `DatasetError`, which are related to the validity of your datasets scientifically (maching conditions and values shape, etc.). The `DatasetError` messages are designed to be clear, so please open an issue on GitHub if there is any confusion.

7.1.4 First-principles phase diagram

By using the `Cu-Mg-input.json` phase description for the fit settings and passing all of the input data in the `input-data` folder, we can first use ESPEI to generate a phase diagram based on single-phase experimental and DFT data. Currently all of the input datasets must be formation properties, and it can be seen that the formation enthalpies are defined from DFT and experiments for the Laves and CuMg2 phases. Mixing enthalpies are defined for the for the fcc, hcp, and Laves phases from DFT and for liquid from experimental measurements.

The following command will generate a database named `cu-mg_dft.tdb` with parameters selected and fit by ESPEI:

```
espei --input espei-in.yaml
```

where `espei-in.yaml` is a *ESPEI input file* with the following contents

```
system:
  phase_models: Cu-Mg-input.json
  datasets: input-data
generate_parameters:
  excess_model: linear
  ref_state: SGTE91
output:
  output_db: cu-mg_dft.tdb
```

The calculation should be relatively quick, on the order of a minute of runtime. With the above command, only mininmal output (warnings) will be reported. You can increase the verbosity to report info messages by setting the `output.verbosity` key to `1` or debug messages with `2`.

With the following code, we can look at the generated phase diagram and compare it to our data.

```
# First-principles phase diagram
from pycalphad import Database, variables as v
from espei.datasets import load_datasets, recursive_glob
from espei.plot import multiplot
import matplotlib.pyplot as plt

# load the experimental and DFT datasets
datasets = load_datasets(recursive_glob('input-data', '*.json'))

# set up the pycalphad phase diagram calculation
dbf = Database('cu-mg_dft.tdb')
comps = ['CU', 'MG', 'VA']
phases = ['LIQUID', 'FCC_A1', 'HCP_A3', 'CUMG2', 'LAVES_C15']
conds = {v.P: 101325, v.T: (300, 1500, 10), v.X('MG'): (0, 1, 0.01)}

# plot the phase diagram and data
multiplot(dbf, comps, phases, conds, datasets)
plt.savefig('cu-mg_dft_phase_diagram.png')
```

Which should result in the following figure

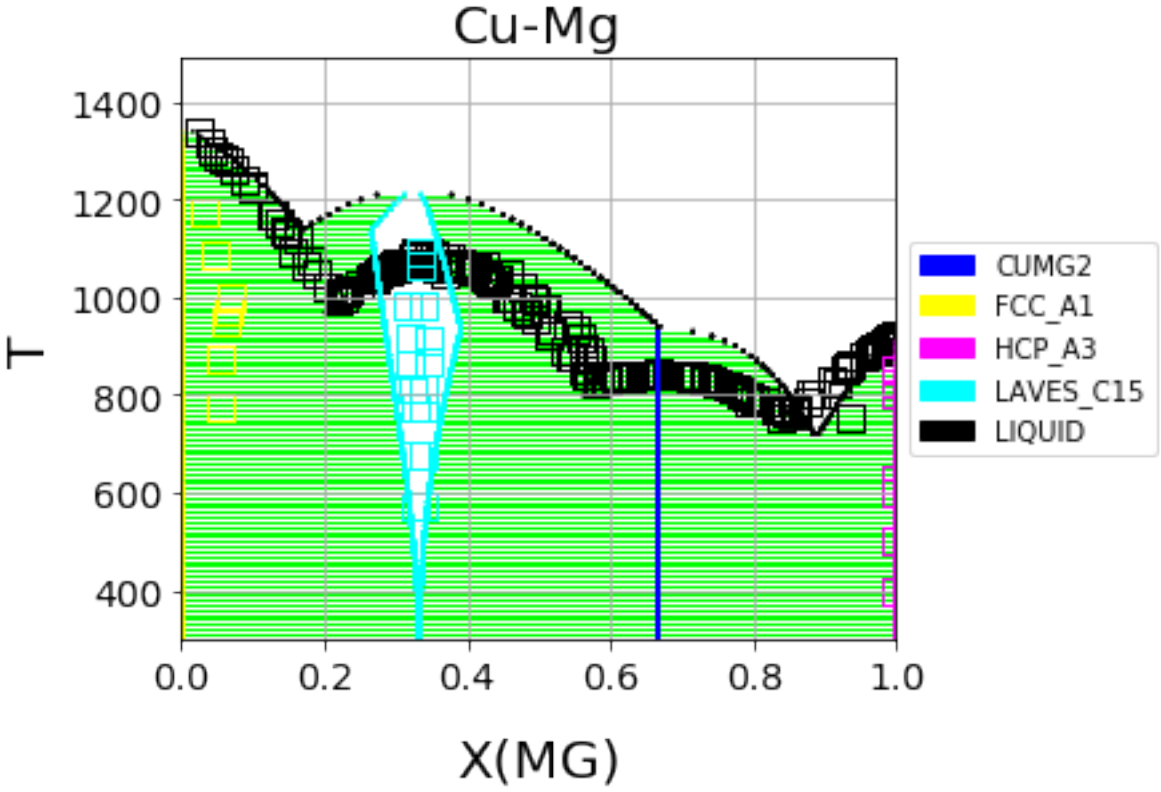


Fig. 1: First-principles Cu-Mg phase diagram from the single-phase fitting in ESPEI

We can see that the phase diagram is already very reasonable compared to the experimental points. The liquidus temperatures and the solubilities of the fcc and Laves phases are the key differences between the equilibrium data and our first-principles phase diagram. The next section will discuss using ESPEI to optimize the parameters selected and calculated based on the single-phase data to these multi-phase equilibria.

7.1.5 MCMC optimization

With the data in the CU-MG input data, ESPEI generated 18 parameters to fit. For systems with more components, solution phases, and input data, may more parameters could be required to describe the thermodynamics of the specific system well. Because they describe Gibbs free energies, parameters in CALPHAD models are highly correlated in both single-phase descriptions and for describing equilibria between phases. For large systems, global numerical optimization of many parameters simultaneously is computationally intractable.

To combat the problem of optimizing many paramters, ESPEI uses MCMC, a stochastic optimzation method.

Now we will use our zero phase fraction equilibria data to optimize our first-principles database with MCMC. The following command will take the database we created in the single-phase parameter selection and perform a MCMC optimization, creating a `cu-mg_mcmc.tdb`:

```
espei --input espei-in.yaml
```

where `espei-in.yaml` is an *ESPEI input file* with the following structure

```
system:
  phase_models: Cu-Mg-input.json
  datasets: input-data
mcmc:
  iterations: 1000
  input_db: cu-mg_dft.tdb
output:
  output_db: cu-mg_mcmc.tdb
```

ESPEI defaults to run 1000 iterations and depends on calculating equilibrium in pycalphad several times for each iteration and the optimization is compute-bound. Fortunately, MCMC optimzations are embarrasingly parallel and ESPEI allows for parallelization using dask or with MPI using mpi4py (single-node only at the time of writing - we are working on it).

Note that you may also see messages about convergence failures or about droppping conditions. These refer to failures to calculate the log-probability or in the pycalphad solver's equilibrium calculation. They are not detrimental to the optimization accuracy, but overall optimization may be slower because those parameter proposals will never be accepted (they return a log-probability of $-\infty$).

Finally, we can use the newly optimized database to plot the phase diagram

```
# Optimized phase diagram from ESPEI's multi-phase fitting
from pycalphad import Database, variables as v
from espei.datasets import load_datasets, recursive_glob
from espei.plot import multiplot
import matplotlib.pyplot as plt

# load the experimental and DFT datasets
datasets = load_datasets(recursive_glob('input-data', '*.json'))

# set up the pycalphad phase diagram calculation
dbf = Database('cu-mg_mcmc.tdb')
comps = ['CU', 'MG', 'VA']
phases = ['LIQUID', 'FCC_A1', 'HCP_A3', 'CUMG2', 'LAVES_C15']
```

(continues on next page)

(continued from previous page)

```
conds = {v.P: 101325, v.T: (300, 1500, 10), v.X('MG'): (0, 1, 0.01)}

# plot the phase diagram and data
multiplot(dbf, comps, phases, conds, datasets)
plt.savefig('cu-mg_mcmc_phase_diagram.png')
```

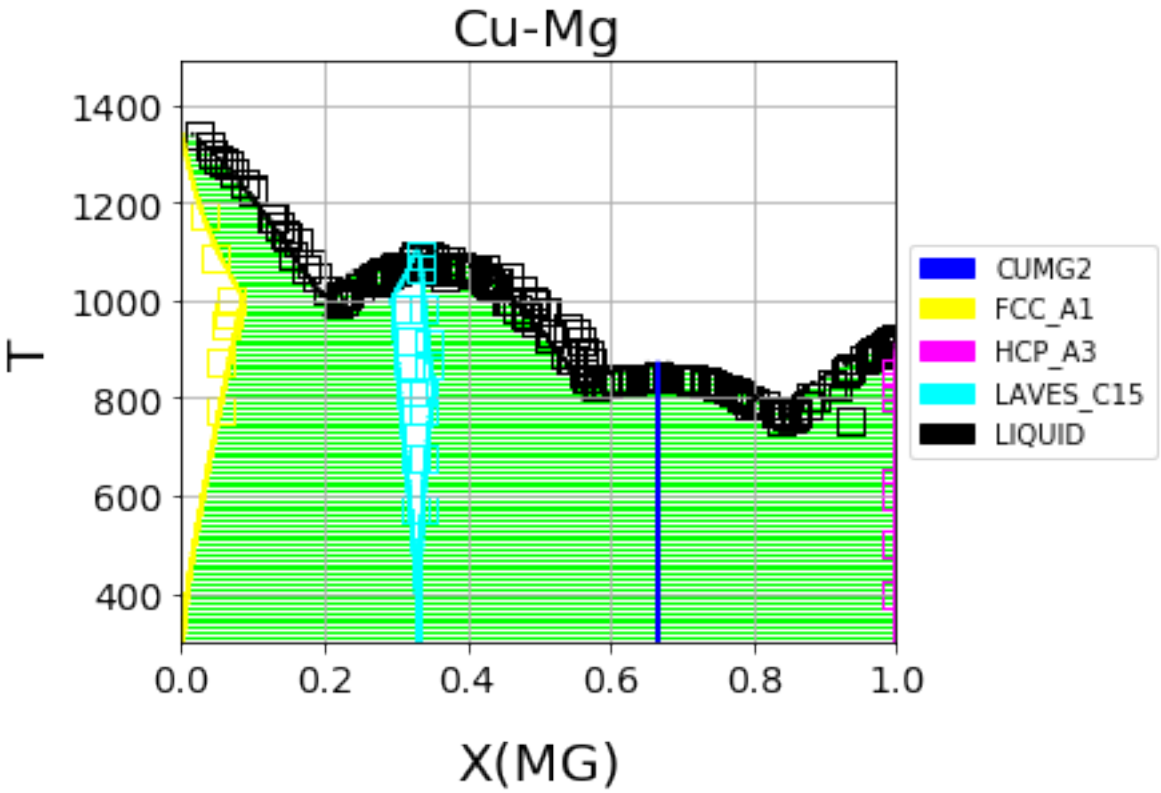


Fig. 2: Optimized Cu-Mg phase diagram from the multi-phase fitting in ESPEI

7.1.6 Analyzing ESPEI Results

After finishing a MCMC run, you will want to analyze your results.

All of the MCMC results will be maintained in two output files, which are serialized NumPy arrays. The file names are set in your `espei-in.yaml` file. The filenames are set by `output.tracefile` and `output.probfile` (documentation) and the defaults are `trace.npy` and `lnprob.npy`, respectively.

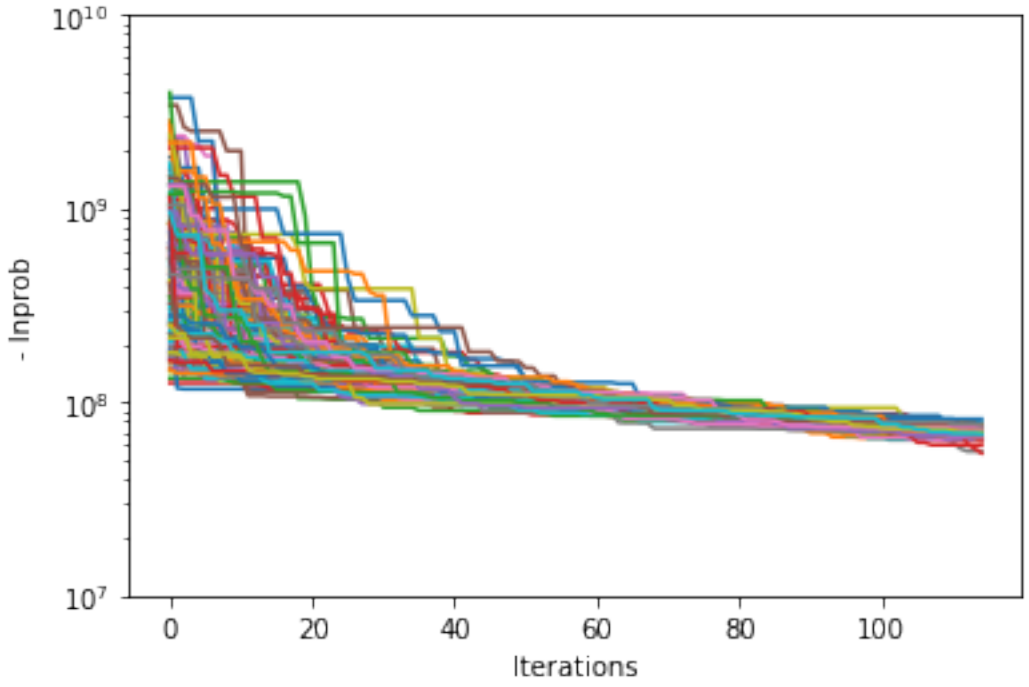
The `tracefile` contains all of the parameters that were proposed over all chains and iterations (the trace). The `probfile` contains all of calculated log probabilities for all chains and iterations (as negative numbers, by convention).

There are several aspects of your data that you may wish to analyze. The next sections will explore some of the options.

Probability convergence

First we'll plot how the probability changes for all of the chains as a function of iterations. This gives a qualitative view of convergence. There are several quantitative metrics that we won't explore here, such as autocorrelation. Qualitatively, this run does not appear converged after 115 iterations.

```
# remove next line if not using iPython or Juypter Notebooks
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from espei.analysis import truncate_arrays

trace = np.load('trace.npy')
lnprob = np.load('lnprob.npy')

trace, lnprob = truncate_arrays(trace, lnprob)


ax = plt.gca()
ax.set_yscale('log')
ax.set_ylim(1e7, 1e10)
ax.set_xlabel('Iterations')
ax.set_ylabel('- lnprob')
num_chains = lnprob.shape[0]
for i in range(num_chains):
    ax.plot(-lnprob[i,:])
plt.show()
```

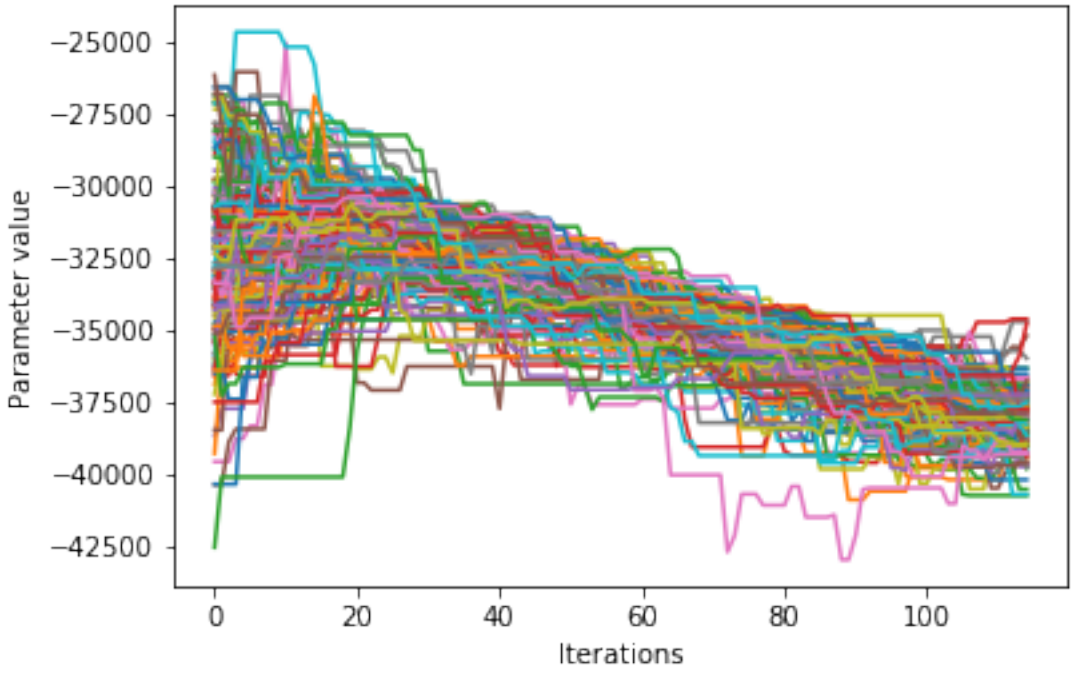


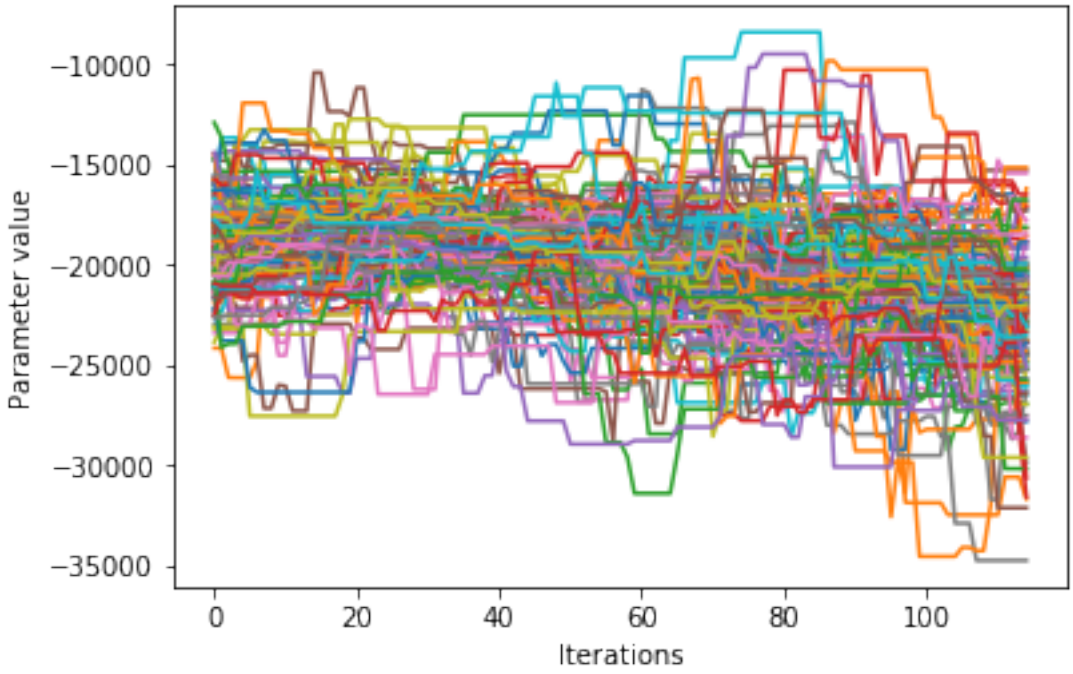
Visualizing the trace of each parameter

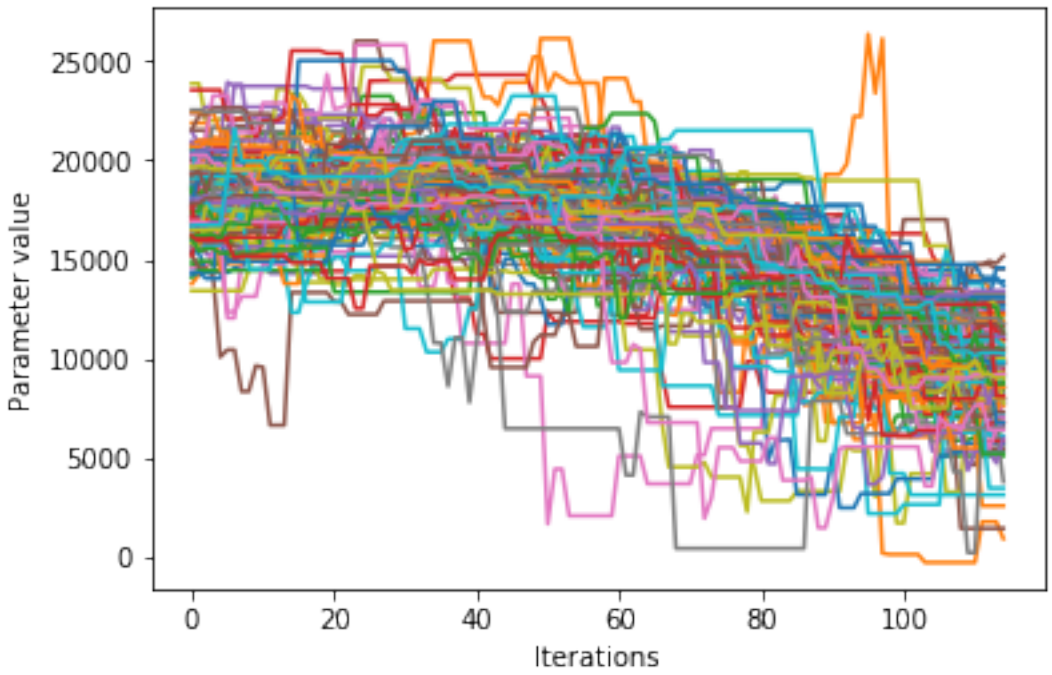
We would like to see how each parameter changed during the iterations. For brevity in the number of plots we'll plot all the chains for each parameter on the same plot. Here we are looking to see how the parameters explore the space

and converge to a solution.

```
# remove next line if not using iPython or Juypter Notebooks
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

from espei.analysis import truncate_arrays

trace = np.load('trace.npy')
lnprob = np.load('lnprob.npy')

trace, lnprob = truncate_arrays(trace, lnprob)

num_chains = trace.shape[0]
num_parameters = 3 # only plot the first three parameter, for all of them use `trace.
↪shape[2]`
for parameter in range(num_parameters):
    ax = plt.figure().gca()
    ax.set_xlabel('Iterations')
    ax.set_ylabel('Parameter value')
    for chain in range(num_chains):
        ax.plot(trace[chain, :, parameter])
plt.show()
```







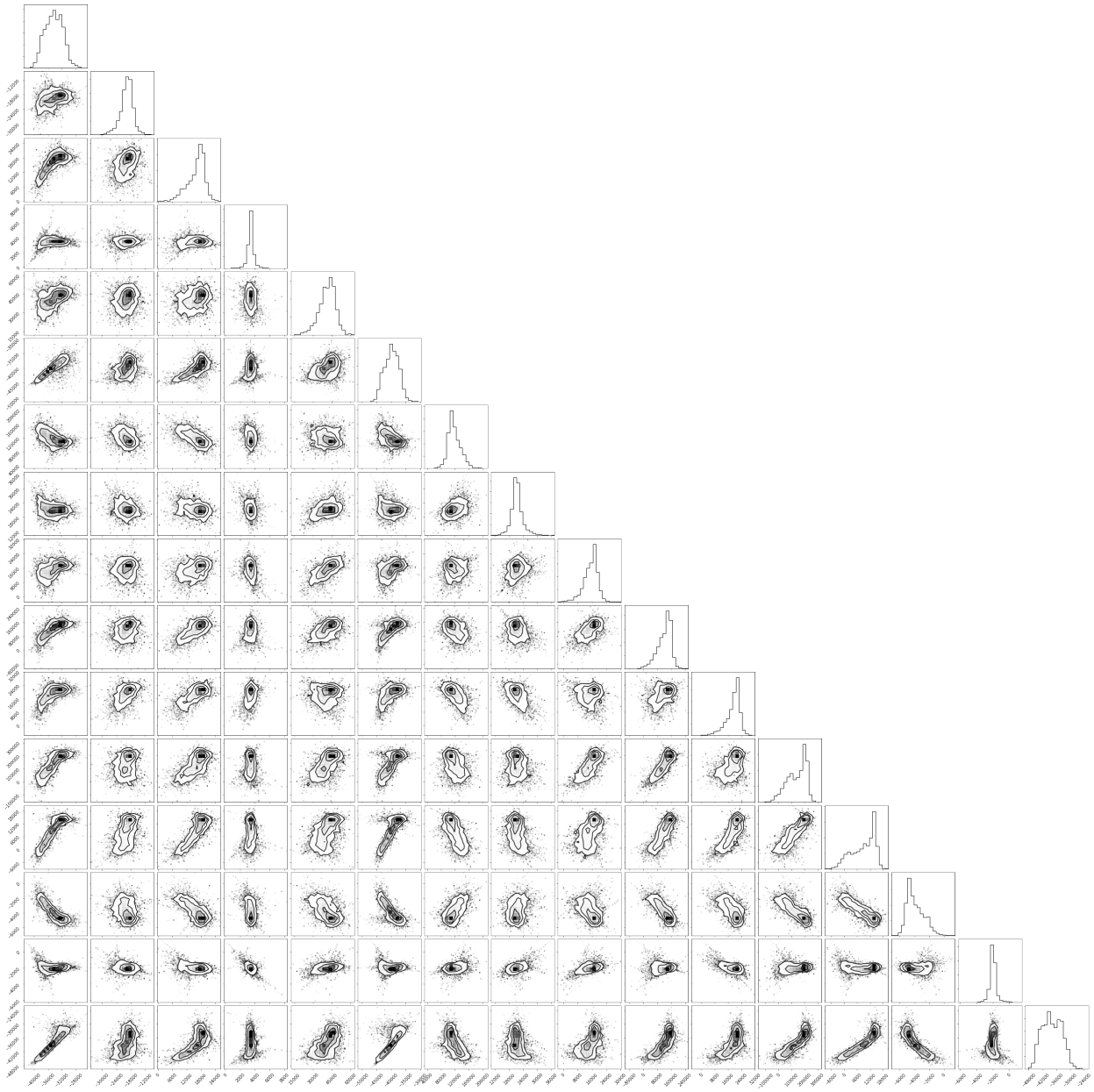
Corner plots

Note: You must install the `corner` package before using it (`conda install corner` or `pip install corner`).

In a corner plot, the distributions for each parameter are plotted along the diagonal and covariances between them under the diagonal. A more circular covariance means that parameters are not correlated to each other, while elongated shapes

indicate that the two parameters are correlated. Strongly correlated parameters are expected for some parameters in CALPHAD models within phases or for phases in equilibrium, because increasing one parameter while decreasing another would give a similar error.

```
# remove next line if not using iPython or Juypter Notebooks
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import corner

from espei.analysis import truncate_arrays

trace = np.load('trace.npy')
lnprob = np.load('lnprob.npy')

trace, lnprob = truncate_arrays(trace, lnprob)

# flatten the along the first dimension containing all the chains in parallel
fig = corner.corner(trace.reshape(-1, trace.shape[-1]))
plt.show()
```



Ultimately, there are many features to explore and we have only covered a few basics. Since all of the results are stored as arrays, you are free to analyze using whatever methods are relevant.

7.1.7 Summary

ESPEI allows thermodynamic databases to be easily reoptimized with little user interaction, so more data can be added later and the database reoptimized at the cost of only computer time. In fact, the existing database from estimates can be used as a starting point, rather than one directly from first-principles, and the database can simply be modified to match any new data.

7.1.8 References

7.1.9 Acknowledgements

Credit for initially preparing the datasets goes to Aleksei Egorov.

CHAPTER 8

Making ESPEI datasets

8.1 Making ESPEI datasets

8.1.1 JSON Format

ESPEI has a single input style in JSON format that is used for all data entry. Single-phase and multi-phase input files are almost identical, but detailed descriptions and key differences can be found in the following sections. For those unfamiliar with JSON, it is fairly similar to Python dictionaries with some rigid requirements

- All string quotes must be double quotes. Use `"key"` instead of `'key'`.
- Numbers should not have leading zeros. `00.123` should be `0.123` and `012.34` must be `12.34`.
- Lists and nested key-value pairs cannot have trailing commas. `{"nums": [1,2,3,],}` is invalid and should be `{"nums": [1,2,3]}`.

These errors can be challenging to track down, particularly if you are only reading the JSON error messages in Python. A visual editor is encouraged for debugging JSON files such as JSONLint. A quick reference to the format can be found at Learn JSON in Y minutes.

ESPEI has support for checking all of your input datasets for errors, which you should always use before you attempt to run ESPEI. This error checking will report all of the errors at once and all errors should be fixed. Errors in the datasets will prevent fitting. To check the datasets at path `my-input-data/` you can run `espei --check-datasets my-input-data`.

8.1.2 Phase Descriptions

The JSON file for describing CALPHAD phases is conceptually similar to a setup file in Thermo-Calc's PARROT module. At the top of the file there is the `refdata` key that describes which reference state you would like to choose. Currently the reference states are strings referring to dictionaries in `pycalphad.refdata` only `"SGTE91"` is implemented.

Each phase is described with the phase name as they key in the dictionary of phases. The details of that phase is a dictionary of values for that key. There are 4 possible entries to describe a phase: `sublattice_model`,

`sublattice_site_ratios`, `equivalent_sublattices`, and `aliases`. `sublattice_model` is a list of lists, where each internal list contains all of the components in that sublattice. The `BCC_B2` sublattice model is `[["AL", "NI", "VA"], ["AL", "NI", "VA"], ["VA"]]`, thus there are three sublattices where the first two have Al, Ni, and vacancies. `sublattice_site_ratios` should be of the same length as the sublattice model (e.g. 3 for `BCC_B2`). The sublattice site ratios can be fractional or integers and do not have to sum to unity.

The optional `equivalent_sublattices` key is a list of lists that describe which sublattices are symmetrically equivalent. Each sub-list in `equivalent_sublattices` describes the indices (zero-indexed) of sublattices that are equivalent. For `BCC_B2` the equivalent sublattices are `[[0, 1]]`, meaning that the sublattice at index 0 and index 1 are equivalent. There can be multiple different sets (multiple sub-lists) of equivalent sublattices and there can be many equivalent sublattices within a sublattice (see `FCC_L12`). If no `equivalent_sublattice` key exists, it is assumed that there are none.a

Finally, the `aliases` key is used to refer to other phases that this sublattice model can describe when symmetry is accounted for. Aliases are used here to describe the `BCC_A2` and `FCC_A1`, which are the disordered phases of `BCC_B2` and `FCC_L12`, respectively. Notice that the aliased phases are not otherwise described in the input file. Multiple phases can exist with aliases to the same phase, e.g. `FCC_L12` and `FCC_L10` can both have `FCC_A1` as an alias.

```
{
  "refdata": "SGTE91",
  "components": ["AL", "NI", "VA"],
  "phases": {
      "LIQUID" : {
      "sublattice_model": [["AL", "NI"]],
      "sublattice_site_ratios": [1]
      },
      "BCC_B2": {
      "aliases": ["BCC_A2"],
      "sublattice_model": [["AL", "NI", "VA"], ["AL", "NI", "VA"], ["VA"]],
      "sublattice_site_ratios": [0.5, 0.5, 1],
      "equivalent_sublattices": [[0, 1]]
      },
      "FCC_L12": {
            "aliases": ["FCC_A1"],
      "sublattice_model": [["AL", "NI"], ["AL", "NI"], ["AL", "NI"], ["AL", "NI"], [
↪"VA"]],
      "sublattice_site_ratios": [0.25, 0.25, 0.25, 0.25, 1],
      "equivalent_sublattices": [[0, 1, 2, 3]]
      },
      "AL3NI1": {
      "sublattice_site_ratios": [0.75, 0.25],
      "sublattice_model": [["AL"], ["NI"]]
      },
      "AL3NI2": {
      "sublattice_site_ratios": [3, 2, 1],
      "sublattice_model": [["AL"], ["AL", "NI"], ["NI", "VA"]]
      },
      "AL3NI5": {
      "sublattice_site_ratios": [0.375, 0.625],
      "sublattice_model": [["AL"], ["NI"]]
      }
    }
}
```

8.1.3 Units

- Energies are in `J/mol-atom` (and the derivatives follow)
- All compositions are mole fractions
- Temperatures are in Kelvin
- Pressures in Pascal

8.1.4 Single-phase Data

Two example of ESPEI input file for single-phase data follow. The first dataset has some data for the formation heat capacity for BCC_B2.

The `components` and `phases` keys simply describe those found in this entry. Use the `reference` key for bookkeeping the source of the data. In `solver` the sublattice configuration and site ratios are described for the phase.

`sublattice_configurations` is a list of different configurations, that should correspond to the sublattices for the phase descriptions. Non-mixing sublattices are represented as a string, while mixing sublattices are represented as a lists. Thus an endmember for `BCC_B2` (as in this example) is `["AL", "NI", VA"]` and if there were mixing (as in the next example) it might be `["AL", ["AL", "NI"], "VA"]`. Mixing also means that the `sublattice_occupancies` key must be specified, but that is not the case in this example. It is important to note that any mixing configurations must have any ideal mixing contributions removed. Regardless of whether there is mixing or not, the length of this list should always equal the number of sublattices in the phase, though the sub-lists can have mixing up to the number of components in that sublattice. Note that the `sublattice_configurations` is a *list* of these lists. That is, there can be multiple sublattice configurations in a single dataset. See the second example in this section for such an example.

The `conditions` describe temperatures (`T`) and pressures (`P`) as either scalars or one-dimensional lists. Most important to describing data are the `output` and `values` keys. The type of quantity is expressed using the `output` key. This can in principle be any thermodynamic quantity, but currently only `CPM*`, `SM*`, and `HM*` (where `*` is either nothing, `_MIX` or `_FORM`) are supported. Support for changing reference states planned but not yet implemented, so all thermodynamic quantities must be formation quantities (e.g. `HM_FORM` or `HM_MIX`, etc.).

The `values` key is the most complicated and care must be taken to avoid mistakes. `values` is a 3-dimensional array where each value is the `output` for a specific condition of pressure, temperature, and sublattice configurations from outside to inside. Alternatively, the size of the array must be `(len(P), len(T), len(subl_config))`. In the example below, the shape of the `values` array is (1, 12, 1) as there is one pressure scalar, one sublattice configuration, and 12 temperatures.

There is also a key to `"excluded_model_contributions"`, which will make those contributions of pycalphad's `Model` not be fit to when doing parameter selection or MCMC. This is useful for cases where the type of data used does not include some specific `Model` contributions that parameters may already exist for. For example, DFT formation energies do not include ideal mixing or (CALPHAD-type) magnetic model contributions, but formation energies from experiments would include these contributions so experimental formation energies should not be excluded. In MCMC, this only takes effect for calculating single phase error (multiphase and activity error do not exclude any model contributions).

```
{
  "reference": "Yi Wang et al 2009",
  "components": ["AL", "NI", "VA"],
  "phases": ["BCC_B2"],
  "solver": {
          "sublattice_site_ratios": [0.5, 0.5, 1],
          "sublattice_configurations": [["AL", "NI", "VA"]],
          "comment": "NiAl sublattice configuration (2SL)"
```

(continues on next page)

(continued from previous page)

```
  },
  "conditions": {
          "P": 101325,
          "T": [  0,  10,  20,  30,  40,  50,  60,  70,  80,  90, 100, 110]
  },
  "excluded_model_contributions": ["idmix", "mag"]
  "output": "CPM_FORM",
  "values":   [[[ 0       ],
                [-0.0173 ],
                [-0.01205],
                [ 0.12915],
                [ 0.24355],
                [ 0.13305],
                [-0.1617 ],
                [-0.51625],
                [-0.841  ],
                [-1.0975 ],
                [-1.28045],
                [-1.3997 ]]]
}
```

In the second example below, there is formation enthalpy data for multiple sublattice configurations. All of the keys and values are conceptually similar. Here, instead of describing how the `output` quantity changes with temperature or pressure, we are instead only comparing `HM_FORM` values for different sublattice configurations. The key differences from the previous example are that there are 9 different sublattice configurations described by `sublattice_configurations` and `sublattice_occupancies`. Note that the `sublattice_configurations` and `sublattice_occupancies` should have exactly the same shape. Sublattices without mixing should have single strings and occupancies of one. Sublattices that do have mixing should have a site ratio for each active component in that sublattice. If the sublattice of a phase is `["AL", "NI", "VA"]`, it should only have two occupancies if only `["AL", "NI"]` are active in the sublattice configuration.

The last difference to note is the shape of the `values` array. Here there is one pressure, one temperature, and 9 sublattice configurations to give a shape of (1, 1, 9).

```
{
  "reference": "C. Jiang 2009 (constrained SQS)",
  "components": ["AL", "NI", "VA"],
  "phases": ["BCC_B2"],
  "solver": {
          "sublattice_occupancies": [
                                     [1, [0.5, 0.5], 1],
                                     [1, [0.75, 0.25], 1],
                                     [1, [0.75, 0.25], 1],
                                     [1, [0.5, 0.5], 1],
                                     [1, [0.5, 0.5], 1],
                                     [1, [0.25, 0.75], 1],
                                     [1, [0.75, 0.25], 1],
                                     [1, [0.5, 0.5], 1],
                                     [1, [0.5, 0.5], 1]
                                    ],
          "sublattice_site_ratios": [0.5, 0.5, 1],
          "sublattice_configurations": [
                                        ["AL", ["NI", "VA"], "VA"],
                                        ["AL", ["NI", "VA"], "VA"],
                                        ["NI", ["AL", "NI"], "VA"],
                                        ["NI", ["AL", "NI"], "VA"],
```

(continues on next page)

(continued from previous page)

```
                                        ["AL", ["AL", "NI"], "VA"],
                                        ["AL", ["AL", "NI"], "VA"],
                                        ["NI", ["AL", "VA"], "VA"],
                                        ["NI", ["AL", "VA"], "VA"],
                                        ["VA", ["AL", "NI"], "VA"]
                                      ],
            "comment": "BCC_B2 sublattice configuration (2SL)"
  },
  "conditions": {
            "P": 101325,
            "T": 300
  },
  "output": "HM_FORM",
  "values":   [[[-40316.61077, -56361.58554,
                 -49636.39281, -32471.25149, -10890.09929,
                 -35190.49282, -38147.99217, -2463.55684,
                 -15183.13371]]]
}
```

8.1.5 Multi-phase Data

The difference between single- and multi-phase is data is in the absence of the `solver` key, since we are no longer concerned with individual site configurations, and the `values` key where we need to represent phase equilibria rather than thermodynamic quantities. Notice that the type of data we are entering in the `output` key is `ZPF` (zero-phase fraction) rather than `CP_FORM` or `H_MIX`. Each entry in the ZPF list is a list of all phases in equilibrium, here `[["AL3NI2", ["NI"], [0.4083]], ["BCC_B2", ["NI"], [0.4340]]]` where each phase entry has the name of the phase, the composition element, and the composition of the tie line point. If there is no corresponding tie line point, such as on a liquidus line, then one of the compositions will be `null`: `[["LIQUID", ["NI"], [0.6992]], ["BCC_B2", ["NI"], [null]]]`. Three- or n-phase equilibria are described as expected: `[["LIQUID", ["NI"], [0.752]], ["BCC_B2", ["NI"], [0.71]], ["FCC_L12", ["NI"], [0.76]]]`.

Note that for higher-order systems the component names and compositions are lists and should be of length `c-1`, where `c` is the number of components.

```
{
  "components": ["AL", "NI"],
  "phases": ["AL3NI2", "BCC_B2"],
  "conditions": {
            "P": 101325,
            "T": [1348, 1176, 977]
  },
  "output": "ZPF",
  "values":   [
           [["AL3NI2", ["NI"], [0.4083]], ["BCC_B2", ["NI"], [0.4340]]],
                [["AL3NI2", ["NI"], [0.4114]], ["BCC_B2", ["NI"], [0.4456]]],
                [["AL3NI2", ["NI"], [0.4114]], ["BCC_B2", ["NI"], [0.4532]]]
              ],
  "reference": "37ALE"
}
```

8.1.6 Activity Data

Activity data is very similar to thermochemical data, except we must enter a reference state. Another minor detail is that that non-endmember compositions must be represented by composition conditions rather than as sublattice occupancies because it's the result of equilibrium calculations where we cannot know the sublattice occupancies. An example for Cu-Mg activties follows, with data digitized from S.P. Garg, Y.J. Bhatt, C. V. Sundaram, Thermodynamic study of liquid Cu-Mg alloys by vapor pressure measurements, Metall. Trans. 4 (1973) 283–289. doi:10.1007/BF02649628.

```
{
  "components": ["CU", "MG", "VA"],
  "phases": ["LIQUID"],
  "solver": {
    "mode": "manual",
    "sublattice_site_ratios": [1],
    "sublattice_configurations": [["CU", "MG"]]
  },
  "reference_state": {
    "phases": ["LIQUID"],
    "conditions": {
      "P": 101325,
      "T": 1200,
      "X_CU": 0.0
    }
  },
  "conditions": {
    "P": 101325,
    "T": 1200,
    "X_CU": [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.0]
  },

  "output": "ACR_MG",
    "values":   [[[0.0057,0.0264,0.0825,0.1812,0.2645,0.4374,0.5852,0.7296,0.882,1.
↪0]]],
  "reference": "garg1973thermodynamic",
  "comment": "Digitized Figure 3 and converted from activity coefficients."
}
```

8.1.7 Tags

Tags are a flexible method to adjust many ESPEI datasets simultaneously and drive them via the ESPEI's input YAML file. Each dataset can have a `"tags"` key, with a corresponding value of a list of tags, e.g. `["dft"]`. Any tag modifications present in the input YAML file are applied to the datasets before ESPEI is run.

They can be used in many creative ways, but some suggested ways include to add weights or to exclude model contributions, e.g. for DFT data that should not have contributions for a CALPHAD magnetic model or ideal mixing energy. An example of using the tags in an input file looks like:

```
{
  "components": ["CR", "FE", "VA"],"phases": ["BCC_A2"],
  "solver": {"mode": "manual", "sublattice_site_ratios": [1, 3],
             "sublattice_configurations": [[["CR", "FE"], "VA"]],
  "sublattice_occupancies": [[[0.5, 0.5], 1.0]]},
  "conditions": {"P": 101325, "T": 300},
  "output": "HM_MIX",
  "values": [[[10000]]],
```

(continues on next page)

(continued from previous page)

```
  "tags": ["dft"]
}
```

An example input YAML looks like

```
system:
  phase_models: CR-FE.json
  datasets: FE-NI-datasets-sep
  tags:
    dft:
      excluded_model_contributions: ["idmix", "mag"]

generate_parameters:
  excess_model: linear
  ref_state: SGTE91
  ridge_alpha: 1.0e-20
output:
  verbosity: 2
  output_db: out.tdb
```

This will add the key `"excluded_model_contributions"` to all datasets that have the `"dft"` tag:

```
{
  "components": ["CR", "FE", "VA"],"phases": ["BCC_A2"],
  "solver": {"mode": "manual", "sublattice_site_ratios": [1, 3],
             "sublattice_configurations": [[["CR", "FE"], "VA"]],
  "sublattice_occupancies": [[[0.5, 0.5], 1.0]]},
  "conditions": {"P": 101325, "T": 300},
  "output": "HM_MIX",
  "values": [[[10000]]],
  "excluded_model_contributions": ["idmix", "mag"]
}
```

8.1.8 Common Mistakes and Notes

1. A single element sublattice is different in a phase model (`[["A", "B"], ["A"]]`) than a sublattice configuration (`[["A", "B"], "A"]`).
2. Make sure you use the right units (`J/mole-atom`, mole fractions, Kelvin, Pascal)
3. Mixing configurations should not have ideal mixing contributions.
4. All types of data can have a `weight` key at the top level that will weight the standard deviation parameter in MCMC runs for that dataset. If a single dataset should have different weights applied, multiple datasets should be created.

Part III

Reference

CHAPTER 9

YAML input files

9.1 ESPEI YAML input files

This page aims to completely describe the ESPEI input file in the YAML format. Possibly useful links are the YAML refcard and the (possibly less useful) Full YAML specification. These are all key value pairs in the format

```
key: value
```

They are nested for purely organizational purposes.

```
top_level_key:
  key: value
```

As long as keys are nested under the correct heading, they have no required order. All of the possible keys are

```
system:
  phase_models
  datasets
  tags

output:
  verbosity
  logfile
  output_db
  tracefile
  probfile

generate_parameters:
  excess_model
  ref_state
  ridge_alpha

mcmc:
  iterations
```

(continues on next page)

(continued from previous page)

```
prior
save_interval
cores
scheduler
input_db
restart_trace
chains_per_parameter
chain_std_deviation
deterministic
data_weights
```

The next sections describe each of the keys individually. If a setting has a default of `required` it must be set explicitly.

9.1.1 system

The `system` key is intended to describe the specific system you are fitting, including the components, phases, and the data to fit to.

phase_models

type string

default required

The JSON file describing the CALPHAD models for each phase. See *Phase Descriptions* for an example of how to write this file.

datasets

type string

default required

The path to a directory containing JSON files of input datasets. The file extension to each of the datasets must be named as `.json`, but they can otherwise be named freely.

For an examples of writing these input JSON files, see *Making ESPEI datasets*.

tags

type dict

default required

Mapping of keys to values to add to datasets with matching tags. These can be used to dynamically drive values in datasets without adjusting the datasets themselves. Useful for adjusting weights or other values in datasets in bulk. For an examples of using tags in input JSON files, see *Tags*.

9.1.2 output

verbosity

type int

default 0

Controls the logging level. Most users will probably want to use `Info` or `Trace`.

`Warning` logs should almost never occur and this log level will be relatively quiet. `Debug` is a fire hose of information, but may be useful in fixing calculation errors or adjusting weights.

Value	Log Level
0	Warning
1	Info
2	Trace
3	Debug

logfile

type string

default null

Name of the file that the logs (controlled by `verbosity`) will be output to. The default is `None` (in Python, `null` in JSON), meaning the logging will be output to stdout and stderr.

output_db

type string

default out.tdb

The database to write out. Can be any file format that can be written by a pycalphad Database.

tracefile

type string

default trace.npy

Name of the file that the MCMC trace is written to. The array has shape `(number of chains, iterations, number of parameters)`.

The array is preallocated and padded with zeros, so if you selected to take 2000 MCMC iterations, but only got through 1500, the last 500 values would be all 0.

You must choose a unique file name. An error will be raised if file specified by `tracefile` already exists. If you don't want a file to be output (e.g. for debugging), you can enter `None`.

probfile

type string

default lnprob.npy

Name of the file that the MCMC ln probabilities are written to. The array has shape `(number of chains, iterations)`.

The array is preallocated and padded with zeros, so if you selected to take 2000 MCMC iterations, but only got through 1500, the last 500 values would be all 0.

You must choose a unique file name. An error will be raised if file specified by `probfile` already exists. If you don't want a file to be output (e.g. for debugging), you can enter `None`.

9.1.3 generate_parameters

The options in `generate_parameters` are used to control parameter selection and fitting to single phase data. This should be used if you have input thermochemical data, such as heat capacities and mixing energies.

Generate parameters will use the Akaike information criterion to select model parameters and fit them, creating a database.

excess_model

type string

default required

options linear

Which type of model to use for excess mixing parameters. Currently only *linear* is supported.

The *exponential* model is planned, as well as support for custom models.

ref_state

type string

default required

options SGTE91 | SR2016

The reference state to use for the pure elements and lattice stabilities. Currently only *SGTE91* and *SR2016* (for certain elements) is supported.

There are plans to extend to support custom reference states.

ridge_alpha

type float

default 1.0e-100

Controls the ridge regression hyperparameter, $ alpha $, as given in the following equation for the ridge regression problem

$$\min_w ||Xw - y||_2^2 + \alpha ||w||_2^2$$

`ridge_alpha` should be a positive floating point number which scales the relative contribution of parameter magnitudes to the residuals.

If an exponential form is used, the floating point value must have a decimal place before the `e`, that is `1e-4` is invalid while `1.e-4` is valid. More generally, the floating point must match the following regular expression per the YAML 1.1 spec: `[-+]?([0-9][0-9_]*)?\.[0-9.]*([eE][-+][0-9]+)?`.

9.1.4 mcmc

The options in `mcmc` control how Markov Chain Monte Carlo is performed using the emcee package.

In order to run an MCMC fitting, you need to specify one and only one source of parameters somewhere in the input file. The parameters can come from including a `generate_parameters` step, or by specifying the `mcmc.input_db` key with a file to load as pycalphad Database.

If you choose to use the parameters from a database, you can then further control settings based on whether it is the first MCMC run for a system (you are starting fresh) or whether you are continuing from a previous run (a 'restart').

iterations

type int

default required

Number of iterations to perform in emcee. Each iteration consists of accepting one step for each chain in the ensemble.

prior

type list or dict

default {'name': 'zero'}

Either a single prior dictionary or a list of prior dictionaries corresponding to the number of parameters. See *Specifying Priors* for examples and details on writing priors.

save_interval

type int

default 1

Controls the interval (in number of iterations) for saving the MCMC chain and probability files. By default, new files will be written out every iteration. For large files (many mcmc iterations and chains per parameter), these might become expensive to write out to disk.

cores

type int

min 1

How many cores from available cores to use during parallelization with dask or emcee. If the chosen number of cores is larger than available, then this value is ignored and espei defaults to using the number available.

Cores does not take affect for MPIPool scheduler option. MPIPool requires the number of processors be set directly with MPI.

scheduler

type string

default dask

options dask | None | JSON file

Which scheduler to use for parallelization. You can choose from either *dask*, *None*, or pass the path to a JSON scheduler file created by dask-distributed.

Choosing dask allows for the choice of cores used through the cores key.

Choosing None will result in no parallel scheduler being used. This is useful for debugging.

Passing the path to a JSON scheduler file will use the resources set up by the scheduler. JSON file schedulers are most useful because schedulers can be started on MPI clusters using `dask-mpi` command. See *Advanced Schedulers* for more information.

input_db

type string

default null

A file path that can be read as a pycalphad Database. The parameters to fit will be taken from this database.

For a parameter to be fit, it must be a symbol where the name starts with `VV`, e.g. `VV0001`. For a `TDB` formatted database, this means that the free parameters must be functions of a single value that are used in your parameters. For example, the following is a valid symbol to fit:

```
FUNCTION VV0000  298.15  10000; 6000 N !
```

restart_trace

type string

default null

If you have run a previous MCMC calculation, then you will have a trace file that describes the position and history of all of the chains from the run. You can use these chains to start the emcee run and pick up from where you left off in the MCMC run by passing the trace file (e.g. `chain.npy`) to this key.

If you are restarting from a previous calculation, you must also specify the same database file (with `input_db`) as you used to run that calculation.

chains_per_parameter

type int

default 2

This controls the number of chains to run in the MCMC calculation as an integer multiple of the number of parameters.

This parameter can only be used when initializing the first MCMC run. If you are restarting a calculation, the number of chains per parameter is fixed by the number you chose previously.

Ensemble samplers require at least `2*p` chains for `p` fitting parameters to be able to make proposals. If `chains_per_parameter = 2`, then the number of chains if there are 10 parameters to fit is 20.

The value of `chains_per_parameter` must be an **EVEN integer**.

chain_std_deviation

type float

default 0.1

The standard deviation to use when initializing chains in a Gaussian distribution from a set of parameters as a fraction of the parameter.

A value of 0.1 means that for parameters with values `(-1.5, 2000, 50000)` the chains will be initialized using those values as the mean and `(0.15, 200, 5000)` as standard deviations for each parameter, respectively.

This parameter can only be used when initializing the first MCMC run. If you are restarting a calculation, the standard deviation for your chains are fixed by the value you chose previously.

You may technically set this to any positive value, you would like. Be warned that too small of a standard deviation may cause convergence to a local minimum in parameter space and slow convergence, while a standard deviation that is too large may cause convergence to meaningless thermodynamic descriptions.

deterministic

type bool

default True

Toggles whether ESPEI runs are deterministic. If this is True, running ESPEI with the same Database and initial settings (either the same `chains_per_parameter` and `chain_std_deviation` or the same `restart_trace`) will result in exactly the same results.

Starting two runs with the same TDB or with parameter generation (which is deterministic) will result in the chains being at exactly the same position after 100 iterations. If these are both restarted after 100 iterations for another 50 iterations, then the final chain after 150 iterations will be the same.

It is important to note that this is only explictly True when *starting* at the same point. If Run 1 and Run 2 are started with the same initial parameters and Run 1 proceeds 50 iterations while Run 2 proceeds 100 iterations, restarting Run 1 for 100 iterations and Run 2 for 50 iterations (so they are both at 150 total iterations) will **NOT** give the same result.

data_weights

type dict

default {'ZPF': 1.0, 'ACR': 1.0, 'HM': 1.0, 'SM': 1.0, 'CPM': 1.0}

Each type of data can be weighted: zero phase fraction (`ZPF`), activity (`ACR`) and the different types of thermochemical error. These weights are used to modify the initial standard deviation of each data type by

$$\sigma = \frac{\sigma_{\text{initial}}}{w}$$

CHAPTER 10

Specifying Priors

10.1 Specifying Priors

In Bayesian statistics, data are used to update prior distributions for all parameters to calculate posterior distributions. A basic introduction to priors and Bayesian statistics can be found in "Kruschke, J. (2014). Doing Bayesian data analysis: A tutorial with R, JAGS, and Stan. Academic Press.". A more advanced treatment is given in "Gelman, A., Stern, H. S., Carlin, J. B., Dunson, D. B., Vehtari, A., & Rubin, D. B. (2013). Bayesian data analysis. Chapman and Hall/CRC.".

ESPEI provides a flexible interface to specify priors your want to use for a variety of parameters of different sign and magnitude through the `espei.priors.PriorSpec` class. This section will cover how to

1. Define flexible, built-in priors using the YAML input and ESPEI script
2. Use custom priors programatically

10.1.1 Built-in Priors

ESPEI has several built-in priors that correspond to functions in `scipy.stats`: `uniform`, `normal`, and `triangular`. There is also a special (improper) `zero` prior that always gives $$ ln p = 0 $$, which is the default.

Each `scipy.stats` prior is typically specified using several keyword argument parameters, e.g. `loc` and `scale`, which have special meaning for the different distribution functions. In order to be flexible to specifying these arguments when the CALPHAD parameters they will be used for are not known beforehand, ESPEI uses a small language to specify how the distribution hyperparameters can be set relative to the CALPHAD parameters.

Basically, the `PriorSpec` objects are created with the name of the distribution and the hyperparameters that are modified with one of the modifier types: `absolute`, `relative`, `shift_absolute`, or `shift_relative`. For example, the `loc` parameter might become `loc_relative` and `scale` might become `scale_shift_relative`.

Here are some examples of how the modifier parameters of value `v` modify the hyperparameters when given a CALPHAD parameter of value `p`:

- `_absolute=v` always take the exact value passed in, `v`; `loc_absolute=-20` gives a value of `loc=-20`.

- `_relative=v` gives , `v*p`; `scale_absolute=0.1` with `p=10000` gives a value of `scale=10000*0.1=1000`.
- `_shift_absolute=v` gives , `p + v`; `scale_shift_absolute=-10` with `p=10000` gives a value of `scale=10000+(-10)=9990`.
- `_shift_relative=v` gives , `p + v*abs(p)`; `loc_shift_relative=-0.5` with `p=-1000` gives a value of `loc=-1000+abs(-1000)*0.5=-1500`.

Note that the hyperparameter prefixes (`loc` or `scale`) are arbitrary and any hyperparameters used in the statistical distributions (e.g. `c` for the triangular distribution) could be used.

YAML

Prior can be specified in the YAML input as a list of dictionaries for different parameters. Since Python objects cannot be instantiated in the YAML files, the `PriorSpec` can be described a dictionary of `{'name': <<NAME>>, 'loc_relative': <<VALUE>>, 'scale_relative': <<VALUE>>, ...}`.

Some common examples in YAML format are as follows:

```
# normal prior, centered on parameter, standard deviation of 0.25*parameter
prior:
  name: normal
  loc_relative: 1.0
  scale_relative: 0.25

# uniform prior from 0 to 2*parameter (or negative(2*parameter) to 0)
prior:
  name: uniform
  loc_shift_relative: -1.0
  scale_relative: 2.0

# triangular prior, centered on parameter, from -0.5*parameter to 0.5*parameter
prior:
  name: triangular
  loc_shift_relative: -0.5
  scale_relative: 1.0
```

Graphically, these are shown below:

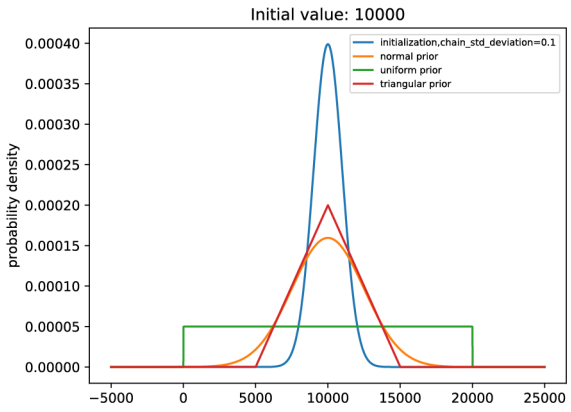


Fig. 1: Example priors compared to initialized parameters.

As a side note: the priors in YAML files can be passed as Python dictionaries, e.g.:

```
# normal prior, centered on parameter, standard deviation of 0.25*parameter
prior: {'name': 'normal', 'loc_relative': 1.0, 'scale_relative': 0.5}
```

Additionally, different priors can be specified using a list of prior specs that match the total degrees of freedom (VV-parameters) in the system. For example, a two parameter system could use a normal and a triangular prior simultaneously:

```
# two priors:
# first a normal prior, centered on parameter, standard deviation of 0.25*parameter
# second a triangular prior, centered on parameter, from -0.5*parameter to 0.
↪5*parameter
prior: [{'name': 'normal', 'loc_relative': 1.0, 'scale_relative': 0.5}, {'name':
↪'triangular', 'loc_shift_relative': -0.5, 'scale_relative': 1.0}]
```

10.1.2 Custom Priors

Generally speaking, a custom prior in ESPEI is any Python object that has a `logpdf` method that takes a parameter and returns the natural log of the probability density function for that parameter. Any distribution you can create using the functions in `scipy.stats`, such as `norm`, is valid.

A list of these custom priors can be passed to ESPEI similar to using built-in priors, but only from the Python interface (not YAML). The number of priors must match the number of parameters, but you can also mix these with the `PriorSpec` objects as desired.

An example of fitting two parameters using a custom gamma distributions with minima at 10 and 100, respectively.

```
from scipy.stats import gamma

my_priors = [gamma(a=1, loc=10), gamma(a=1, loc=100)]

from espei.espei_script import get_run_settings, run_espei


input_dict = {
    'system': {
        'phase_models': 'phases.json',
        'datasets': 'input-data',
    },
    'mcmc': {
        'iterations': '1000',
        'input_db': 'param_gen.tdb',  # must have two parameters to fit
        'prior': my_priors,
    },
}

run_espei(get_run_settings(input_dict))
```

CHAPTER 11

Recipes

11.1 Recipes

Here you can find some useful snippets of code to make using ESPEI easier.

11.1.1 Optimal parameter TDBs

Creating TDBs of optimal parameters from a tracefile and probfile:

```
"""
This script updates an input TDB file with the optimal parameters from an ESPEI run.

Change the capitalized variables to your desired input and output filenames.
"""

INPUT_TDB_FILENAME = 'CU-MG_param_gen.tdb'
OUTPUT_TDB_FILENAME = 'CU-MG_opt_params.tdb'
TRACE_FILENAME = 'trace.npy'
LNPROB_FILENAME = 'lnprob.npy'

import numpy as np
from pycalphad import Database
from espei.analysis import truncate_arrays
from espei.utils import database_symbols_to_fit, optimal_parameters

trace = np.load(TRACE_FILENAME)
lnprob = np.load(LNPROB_FILENAME)
trace, lnprob = truncate_arrays(trace, lnprob)

dbf = Database(INPUT_TDB_FILENAME)
opt_params = dict(zip(database_symbols_to_fit(dbf), optimal_parameters(trace,␣
↪lnprob)))
```

(continues on next page)

(continued from previous page)

```
dbf.symbols.update(opt_params)
dbf.to_file(OUTPUT_TDB_FILENAME)
```

11.1.2 Plotting phase equilibria data

When compiling ESPEI datasets of phase equilibria data, it can be useful to plot the data to check that it matches visually with what you are expecting. This script plots a binary phase diagram.

TIP: Using this in Jupyter Notebooks make it really fast to update and watch your progress.

```
"""
This script will create a plot in a couple seconds from a datasets directory
that you can use to check your phase equilibria data.

Change the capitalized variables to the system information and the
directory of datasets you want to plot.
"""

COMPONENTS = ['CU', 'MG', 'VA']
INDEPENDENT_COMPONENT = "MG"  # component to plot on the x-axis
PHASES = ['BCC_A2', 'CUMG2', 'FCC_A1', 'LAVES_C15', 'LIQUID']

DATASETS_DIRECTORY = "~/my-datasets/CU-MG"

X_MIN, X_MAX = 0.0, 1.0
Y_MIN, Y_MAX = 400, 1400

# script starts here, you shouldn't have to edit below this line
import os
from espei.plot import dataplot
from espei.datasets import recursive_glob, load_datasets
from pycalphad import variables as v
import matplotlib.pyplot as plt

plt.figure(figsize=(10,8))

ds = load_datasets(recursive_glob(os.path.expanduser(DATASETS_DIRECTORY), '*.json'))
conds = {v.P: 101325, v.T: (1,1,1), v.X(INDEPENDENT_COMPONENT): (1, 1, 1)}
dataplot(COMPONENTS, PHASES, conds, ds)
plt.xlim(X_MIN, X_MAX)
plt.ylim(Y_MIN, Y_MAX)
plt.show()
```

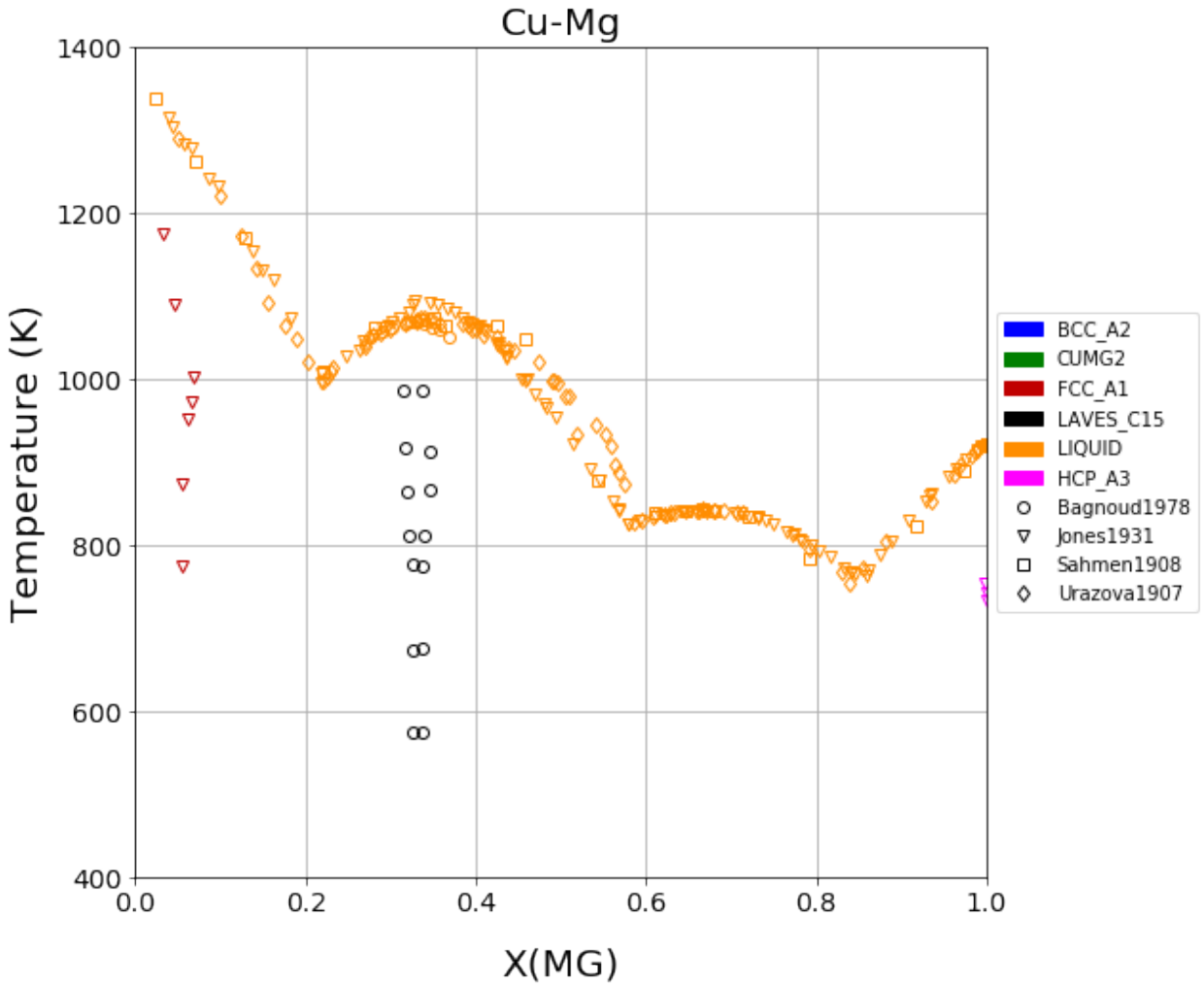
The script gives the following output:

11.1.3 Plotting thermochemical properties parameters with data

Parameter selection in ESPEI fits Calphad parameters to thermochemical data. MCMC can adjust these parameters. In both cases, it may be useful to compare the energies of specific endmembers and interactions to the model. The code below compares the energies for an endmember or interaction (a configuration). The `plot_parameters` code will automatically plot all of the energies that data exists for, but no more.

```
"""
This script plots a single interaction in a database compared to data.
```

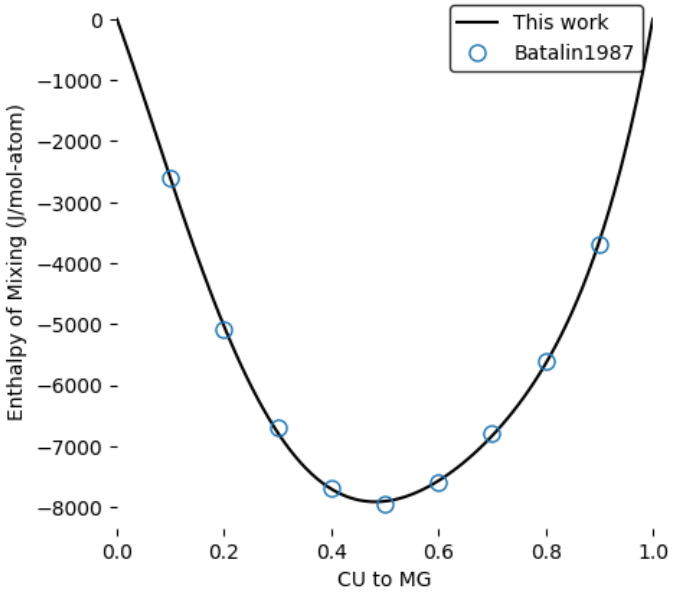
(continues on next page)


Cu-Mg
Temperature (K)
X(MG)
1400
1200
1000
800
600
400
0.0
0.2
0.4
0.6
0.8
1.0
BCC_A2
CUMG2
FCC_A1
LAVES_C15
LIQUID
HCP_A3
Bagnoud1978
Jones1931
Sahmen1908
Urazova1907

(continued from previous page)

```
"""

# Settings
INPUT_TDB_FILENAME = 'CU-MG_param_gen.tdb'
DATASET_DIRECTORY = 'input-data'
PHASE_NAME = 'LIQUID'
# CONFIGURATION must be a tuple of the configuration to be plotted.
# This can only plot one endmember or interaction at a time.
# Note that the outside tuples are the whole configuration
# and the insides are for each individual sublattice.
# Single sublattices *MUST* have the comma after the
# object in order to be a tuple, not just parantheses.
# some examples:
# ('CU', 'MG')  # endmember
# (('CU', 'MG'),)  # (('CU', 'MG')) is invalid because it will be come ('CU', 'MG')
# ('MG', ('CU', 'MG'))
CONFIGURATION = (('CU', 'MG'),)

# Plot the parameter
import matplotlib.pyplot as plt
from pycalphad import Database
from espei.datasets import load_datasets, recursive_glob
from espei.plot import plot_parameters

dbf = Database(INPUT_TDB_FILENAME)
comps = sorted(dbf.elements)
ds = load_datasets(recursive_glob(DATASET_DIRECTORY, '*.json'))
plot_parameters(dbf, comps, PHASE_NAME, CONFIGURATION, datasets=ds, symmetry=None)
plt.show()
```

Running for the single sublattice LIQUID phase in Cu-Mg gives the following output after parameter selection:



CHAPTER 12

Advanced Schedulers

12.1 Advanced Schedulers

ESPEI uses dask-distributed for parallelization and provides an easy way to deploy clusters locally via TCP with the `mcmc.scheduler: dask` setting.

Sometimes ESPEI's dask scheduling options are not sufficiently flexible for different environments.

As an alternative to setting the cores with the `mcmc.scheduler: dask` setting, you can provide ESPEI with a scheduler file from dask that has information about how to connect to a dask parallel scheduler.

This is generally a two step process of

1. Starting a scheduler with workers and writing a scheduler file
2. Running ESPEI and connecting to the existing scheduler

In order to let the system manage the memory and prevent dask from pausing or killing workers, the memory limit should be set to zero.

12.1.1 Starting a scheduler

MPI-based dask scheduler

Dask provides a `dask-mpi` package that sets this up for you and creates a scheduler file to pass to ESPEI. The scheduler information will be serialized as a JSON file that you set in your ESPEI input file.

The dask-mpi package (version 2.0.0 or greater) must be installed before you can use it:

```
conda install -c conda-forge --yes "dask-mpi>=2"
```

Note that you may also need a particular MPI implementation, conda-forge provides packages for OpenMPI or MPICH. You can pick a particular one by installing dask-mpi using either:

```
conda install -c conda-forge --yes "dask-mpi>=2" "mpi=*=openmpi"
```

or

```
conda install -c conda-forge --yes "dask-mpi>=2" "mpi=*=mpich"
```

or let conda pick one for you by not including any.

To start the scheduler and workers in the background, you can run the `dask-mpi` command (use `dask-mpi --help` to check the arguments). The following command will start a scheduler on the main MPI task, then a worker for each remaining MPI task that `mpirun` sees.

```
mpirun dask-mpi --scheduler-file my_scheduler.json --nthreads 1 --memory-limit 0 &
```

Generic scheduler

If you need further customization of dask schedulers, you can start a distributed Client any way you like, then write out the scheduler file for ESPEI to use.

For example, if you name the following file `start_scheduler.py`, you can run this Python script in the background, which will contain the scheduler and workers, then ESPEI will connect to it.

```
# start_scheduler.py
from distributed import Client, LocalCluster
from tornado.ioloop import IOLoop

if __name__ == '__main__':
    loop = IOLoop()
    cluster = LocalCluster(n_workers=4, threads_per_worker=1, memory_limit=0)
    client = Client(cluster)
    client.write_scheduler_file('my-scheduler.json')
    loop.start()  # keeps the scheduler running
    loop.close()
```

Running `start_scheduler.py &`, will run this process in the background with 4 processes.

12.1.2 ESPEI Input

After starting the scheduler on the cluster, you run ESPEI like normal.

For the most part, this ESPEI input file is the same as you use locally, except the `scheduler` parameter is set to the name of your scheduler file.

Here is an example for multiphase fitting starting from a generated TDB with a scheduler file named `my-scheduler.json`:

```
system:
  phase_models: my-phases.json
  datasets: my-input-data
mcmc:
  iterations: 1000
  input_db: my-tdb.tdb
  scheduler: my-scheduler.json
```

12.1.3 Example Queue Script - MPI

To run on through a queueing system, you'll often use queue scripts that start batch jobs.

This example will create an MPI scheduler using `dask-mpi` via `mpirun` (or other MPI executable). Since many MPI jobs are run through batch schedulers, an example script for a PBS job looks like:

```
#!/bin/bash

#PBS -l nodes=1:ppn=20
#PBS -l walltime=48:00:00
#PBS -A open
#PBS -N espei-mpi
#PBS -o espei-mpi.out
#PBS -e espei-mpi.error

# starts the scheduler on MPI and creates the scheduler file called 'my_scheduler.json
↪'
# you can replace this line with any script that starts a scheduler
# e.g. a `start_scheduler.py` file
# make sure it ends with `&` to run the process in the background
mpirun dask-mpi --scheduler-file my_scheduler.json --nthreads 1 --memory-limit 0 &

# runs ESPEI as normal
espei --in espei-mpi-input.yaml
```

12.1.4 References

See http://distributed.readthedocs.io/en/latest/setup.html?highlight=dask-mpi#using-mpi for more details.

12.2 API Documentation

12.2.1 espei package

Subpackages

espei.error_functions package

Submodules

espei.error_functions.activity_error module

Calculate error due to measured activities.

espei.error_functions.activity_error.**calculate_activity_error**(*dbf*, *comps*, *phases*, *datasets*, *parameters=None*, *phase_models=None*, *callables=None*, *data_weight=1.0*)

Return the sum of square error from activity data

Parameters

- **dbf** (*pycalphad.Database*) – Database to consider
- **comps** (*list*) – List of active component names
- **phases** (*list*) – List of phases to consider
- **datasets** (*espei.utils.PickleableTinyDB*) – Datasets that contain single phase data
- **parameters** (*dict*) – Dictionary of symbols that will be overridden in pycalphad.equilibrium
- **phase_models** (*dict*) – Phase models to pass to pycalphad calculations
- **callables** (*dict*) – Callables to pass to pycalphad
- **data_weight** (*float*) – Weight for standard deviation of activity measurements, dimensionless. Corresponds to the standard deviation of differences in chemical potential in typical measurements of activity, in J/mol.

Returns A single float of the sum of square errors

Return type float

Notes

General procedure: 1. Get the datasets 2. For each dataset

a. Calculate reference state equilibrium

b. Calculate current chemical potentials

c. Find the target chemical potentials

d. Calculate error due to chemical potentials

espei.error_functions.activity_error.**chempot_error**(*sample_chempots*, *target_chempots*, *std_dev=10.0*)

Return the sum of square error from chemical potentials

sample_chempots [numpy.ndarray] Calculated chemical potentials

target_activity [numpy.ndarray] Chemical potentials to target

std_dev [float] Standard deviation of activity measurements in J/mol. Corresponds to the standard deviation of differences in chemical potential in typical measurements of activity.

Returns Error due to chemical potentials

Return type float

espei.error_functions.activity_error.**target_chempots_from_activity**(*component*, *target_activity*, *temperatures*, *reference_result*)

Return an array of experimental chemical potentials for the component

Parameters

- **component** (*str*) – Name of the component
- **target_activity** (*numpy.ndarray*) – Array of experimental activities
- **temperatures** (*numpy.ndarray*) – Ravelled array of temperatures (of same size as exp_activity).
- **reference_result** (*xarray.Dataset*) – Dataset of the equilibrium reference state. Should contain a singe point calculation.

Returns Array of experimental chemical potentials

Return type numpy.ndarray

espei.error_functions.context module

Convenience function to create a context for the built in error functions

espei.error_functions.context.**setup_context**(*dbf*, *datasets*, *symbols_to_fit=None*, *data_weights=None*, *make_callables=True*)

Set up a context dictionary for calculating error.

Parameters

- **dbf** (*Database*) – A pycalphad Database that will be fit
- **datasets** (*PickleableTinyDB*) – A database of single- and multi-phase data to fit
- **symbols_to_fit** (*list of str*) – List of symbols in the Database that will be fit. If None (default) are passed, then all parameters prefixed with *VV* followed by a number, e.g. VV0001 will be fit.

Notes

A copy of the Database is made and used in the context. To commit changes back to the original database, the dbf.symbols.update method should be used.

espei.error_functions.thermochemical_error module

Calculate error due to thermochemical quantities: heat capacity, entropy, enthalpy.

espei.error_functions.thermochemical_error.**calculate_points_array**(*phase_constituents*, *configuration*, *occupancies=None*)

Calculate the points array to use in pycalphad calculate calls.

Converts the configuration data (and occupancies for mixing data) into the points array by looking up the indices in the active phase constituents.

Parameters

- **phase_constituents** (*list*) – List of active constituents in a phase
- **configuration** (*list*) – List of the sublattice configuration
- **occupancies** (*list*) – List of sublattice occupancies. Required for mixing sublattices, otherwise takes no effect.

Returns

Return type numpy.ndarray

Notes

Errors will be raised if components in the configuration are not in the corresponding phase constituents sublattice.

espei.error_functions.thermochemical_error.**calculate_thermochemical_error**(*dbf*, *comps*, *thermochemical_data*, *parameters=None*)

Calculate the weighted single phase error in the Database

Parameters

- **dbf** (*pycalphad.Database*) – Database to consider
- **comps** (*list*) – List of active component names
- **thermochemical_data** (*list*) – List of thermochemical data dicts
- **parameters** (*dict*) – Dictionary of symbols that will be overridden in pycalphad.calculate

Returns A single float of the residual sum of square errors

Return type float

Notes

There are different single phase values, HM_MIX, SM_FORM, CP_FORM, etc. Each of these have different units and the error cannot be compared directly. To normalize all of the errors, a normalization factor must be used. Equation 2.59 and 2.60 in Lukas, Fries, and Sundman "Computational Thermodynamics" shows how this can be considered. Each type of error will be weighted by the reciprocal of the estimated uncertainty in the measured value and conditions. The weighting factor is calculated by $ p_i = (Delta L_i)^{-1} $ where $ Delta L_i $ is the uncertainty in the measurement. We will neglect the uncertainty for quantities such as temperature, assuming they are small.

espei.error_functions.thermochemical_error.**get_prop_samples**(*dbf*, *comps*, *phase_name*, *desired_data*)

Return data values and the conditions to calculate them by pycalphad calculate from the datasets

Parameters

- **dbf** (*pycalphad.Database*) – Database to consider
- **comps** (*list*) – List of active component names
- **phase_name** (*str*) – Name of the phase to consider from the Database
- **desired_data** (*list*) – List of dictionary datasets that contain the values to sample

Returns Dictionary of condition kwargs for pycalphad's calculate and the expected values

Return type dict

espei.error_functions.thermochemical_error.**get_thermochemical_data**(*dbf*, *comps*, *phases*, *datasets*, *weight_dict=None*, *symbols_to_fit=None*, *make_callables=True*)

Parameters

- **dbf** (*pycalphad.Database*) – Database to consider
- **comps** (*list*) – List of active component names
- **phases** (*list*) – List of phases to consider
- **datasets** (*espei.utils.PickleableTinyDB*) – Datasets that contain single phase data
- **weight_dict** (*dict*) – Dictionary of weights for each data type, e.g. {'HM': 200, 'SM': 2}
- **symbols_to_fit** (*list*) – Parameters to fit. Used to build the models and callables.

Returns List of data dictionaries to iterate over

Return type list

espei.error_functions.zpf_error module

Calculate driving_force due to ZPF tielines.

The general approach is similar to the PanOptimizer rough search method.

1. With all phases active, calculate the chemical potentials of the tieline endpoints via `equilibrium` calls. Done in `estimate_hyperplane`.
2. Calculate the target chemical potentials, which are the average chemical potentials of all of the current chemical potentials at the tieline endpoints.
3. Calculate the current chemical potentials of the desired single phases
4. The error is the difference between these chemical potentials

There's some special handling for tieline endpoints where we do not know the composition conditions to calculate chemical potentials at.

espei.error_functions.zpf_error.**calculate_zpf_error**(*dbf*, *phases*, *zpf_data*, *phase_models=None*, *parameters=None*, *callables=None*, *data_weight=1.0*)

Calculate error due to phase equilibria data

Parameters

- **dbf** (*pycalphad.Database*) – Database to consider
- **phases** (*list*) – List of phases to consider

- **zpf_data** (*list*) – Datasets that contain single phase data
- **phase_models** (*dict*) – Phase models to pass to pycalphad calculations
- **parameters** (*dict*) – Dictionary of symbols that will be overridden in pycalphad.equilibrium
- **callables** (*dict*) – Callables to pass to pycalphad
- **data_weight** (*float*) – Scaling factor for the standard deviation of the measurement of a tieline which has units J/mol. The standard deviation is 1000 J/mol and the scaling factor defaults to 1.0.

Returns Log probability of ZPF error

Return type float

Notes

The physical picture of the standard deviation is that we've measured a ZPF line. That line corresponds to some equilibrium chemical potentials. The standard deviation is the standard deviation of those 'measured' chemical potentials.

espei.error_functions.zpf_error.**driving_force_to_hyperplane**(*dbf*, *comps*, *current_phase*, *cond_dict*, *target_hyperplane_chempots*, *phase_flag*, *phase_models*, *parameters*, *callables=None*)

Calculate the integrated driving force between the current hyperplane and target hyperplane.

Parameters

- **dbf** (*pycalphad.Database*) – Database to consider
- **comps** (*list*) – List of active component names
- **current_phase** (*list*) – List of phases to consider
- **phase_models** (*dict*) – Phase models to pass to pycalphad calculations
- **parameters** (*dict*) – Dictionary of symbols that will be overridden in pycalphad.equilibrium
- **callables** (*dict*) – Callables to pass to pycalphad
- **cond_dict** (*dict*) – Dictionary of state variables, e.g. v.P and v.T, v.X
- **target_hyperplane_chempots** (*numpy.ndarray*) – Array of chemical potentials for target equilibrium hyperplane.
- **phase_flag** (*str*) – String of phase flag, e.g. 'disordered'.
- **phase_models** – Phase models to pass to pycalphad calculations
- **parameters** – Dictionary of symbols that will be overridden in pycalphad.equilibrium

Returns Single value for the total error between the current hyperplane and target hyperplane.

Return type float

espei.error_functions.zpf_error.**estimate_hyperplane**(*dbf*, *comps*, *phases*, *current_statevars*, *comp_dicts*, *phase_models*, *parameters*, *callables=None*)

Calculate the chemical potentials for the target hyperplane, one vertex at a time

Parameters

- **dbf** (*pycalphad.Database*) – Database to consider
- **comps** (*list*) – List of active component names
- **phases** (*list*) – List of phases to consider
- **current_statevars** (*dict*) – Dictionary of state variables, e.g. v.P and v.T, no compositions.
- **comp_dicts** (*list*) – List of tuples of composition dictionaries and phase flags. Composition dictionaries are pycalphad variable dicts and the flag is a string e.g. ({v.X('CU'): 0.5}, 'disordered')
- **phase_models** (*dict*) – Phase models to pass to pycalphad calculations
- **parameters** (*dict*) – Dictionary of symbols that will be overridden in pycalphad.equilibrium
- **callables** (*dict*) – Callables to pass to pycalphad

Returns Array of chemical potentials.

Return type numpy.ndarray

Notes

This takes just *one* set of phase equilibria, e.g. a dataset point of [['FCC_A1', ['CU'], [0.1]], ['LAVES_C15', ['CU'], [0.3]]] and calculates the chemical potentials given all the phases possible at the given compositions. Then the average chemical potentials of each end point are taken as the target hyperplane for the given equilibria.

espei.error_functions.zpf_error.**get_zpf_data**(*comps*, *phases*, *datasets*)

Return the ZPF data used in the calculation of ZPF error

Parameters

- **comps** (*list*) – List of active component names
- **phases** (*list*) – List of phases to consider
- **datasets** (*espei.utils.PickleableTinyDB*) – Datasets that contain single phase data

Returns List of data dictionaries with keys `weight`, `data_comps` and `phase_regions`. `data_comps` are the components for the data in question. `phase_regions` are the ZPF phases, state variables and compositions.

Return type list

Module contents

Functions for calculating error.

espei.optimizers package

Submodules

espei.optimizers.graph module

Defines a OptNode and OptGraph to be used by OptimizerBase subclasses. Together they define the path of one or more optimizations and can be used to store and replay optimization history.

class espei.optimizers.graph.**OptGraph**(*root*)
Bases: object

Directed acyclic graph of optimal parameters.

Notes

The OptGraph defines a directed acyclic graph of commits. Each commit corresponds to a single OptNode. The root node is intended to be the fresh parameters from the database before any optimization. Therefore, any path from the root node to any other node represents a set of optimizations to the parameters in the database.

add_node(*node*, *parent*)

static get_path_to_node(*node*)
Return the path from the root to the node.

Parameters **node** (OptNode) –

Returns

Return type list of OptNode

get_transformation_dict(*node*)
Return a dictionary of parameters from the path walked from the root to the passed node.

Parameters **node** (OptNode) –

Returns

Return type dict

class espei.optimizers.graph.**OptNode**(*parameters*, *datasets*, *node_id=None*)
Bases: object

Node as the result of an optimization.

parameters

Type dict

datasets

Type PickleableTinyDB

id

Type int

parent

Type *OptNode*

children

Type list of OptNode

Notes

OptNodes are individual nodes in the graph that correspond to the result of a call to fit - they represent optimized parameters given the parent state and some data (also part of the OptNode).

Each OptNode can only be derived from one set of parameters, however one parameter state may be a branching point to many new parameter states, so an OptNode can have only one parent, but many children.

espei.optimizers.opt_base module

class espei.optimizers.opt_base.**OptimizerBase**(*dbf*)
Bases: object

Enable fitting and replaying fitting steps

commit()

discard()
Discard all staged nodes

fit(*symbols*, *datasets*, **args*, ***kwargs*)

static predict(*params*, *context*)
Given a set of parameters and a context, return the resulting sum of square error.

Parameters

- **params** (*list*) – 1 dimensional array of parameters
- **context** (*dict*) – Dictionary of arguments/keyword arguments to pass to functions

Returns

Return type float

reset_database()
Set the Database to the state of the current node

espei.optimizers.opt_mcmc module

class espei.optimizers.opt_mcmc.**EmceeOptimizer**(*dbf*, *scheduler=None*)
Bases: *espei.optimizers.opt_base.OptimizerBase*

An optimizer using an EnsembleSampler based on Goodman and Weare [1] implemented in emcee [2]

scheduler
An object implementing a *map* function

Type mappable

save_interval
Interval of iterations to save the tracefile and probfile.

Type int

tracefile
Filename to store the trace with NumPy.save. Array has shape (chains, iterations, parameters). Defaults to None.

Type str

probfile
filename to store the log probability with NumPy.save. Has shape (chains, iterations)

Type str

References

[1] Goodman and Weare, Ensemble Samplers with Affine Invariance. Commun. Appl. Math. Comput. Sci. 5, 65-80 (2010). [2] Foreman-Mackey, Hogg, Lang, Goodman, emcee: The MCMC Hammer. Publ. Astron. Soc. Pac. 125, 306-312 (2013).

do_sampling(*chains*, *iterations*)

static get_priors(*prior*, *symbols*, *params*)
Build priors for a particular set of fitting symbols and initial parameters. Returns a dict that should be used to update the context.

Parameters

- **prior** (*dict or PriorSpec or None*) – Prior to initialize. See the docs on
- **symbols** (*list of str*) – List of symbols that will be fit
- **params** (*list of float*) – List of parameter values corresponding to the symbols. These should be the initial parameters that the priors will be based off of.

static initialize_chains_from_trace(*restart_trace*)

static initialize_new_chains(*params*, *chains_per_parameter*, *std_deviation*, *deterministic=True*)
Return an array of num_samples from a Gaussian distribution about each parameter.

Parameters

- **params** (*ndarray*) – 1D array of initial parameters that will be the mean of the distribution.
- **num_samples** (*int*) – Number of chains to initialize.
- **chains_per_parameter** (*int*) – number of chains for each parameter. Must be an even integer greater or equal to 2. Defaults to 2.
- **std_deviation** (*float*) – Fractional standard deviation of the parameters to use for initialization.
- **deterministic** (*bool*) – True if the parameters should be generated deterministically.

Returns

Return type ndarray

static predict(*params*, ***ctx*)
Calculate lnprob = lnlike + lnprior

save_sampler_state()
Convenience function that saves the trace and lnprob if they haven't been set to None by the user.

Requires that the sampler attribute be set.

espei.optimizers.opt_scipy module

class espei.optimizers.opt_scipy.**SciPyOptimizer**(*dbf*)
Bases: *espei.optimizers.opt_base.OptimizerBase*

static predict(*params*, *ctx*)
Given a set of parameters and a context, return the resulting sum of square error.

Parameters

- **params** (*list*) – 1 dimensional array of parameters
- **context** (*dict*) – Dictionary of arguments/keyword arguments to pass to functions

Returns

Return type float

espei.optimizers.utils module

exception espei.optimizers.utils.**OptimizerError**
Bases: BaseException

Module contents

espei.parameter_selection package

Submodules

espei.parameter_selection.model_building module

Building candidate models

espei.parameter_selection.model_building.**build_candidate_models**(*configuration*, *features*)
Return a dictionary of features and candidate models

Parameters

- **configuration** (*tuple*) – Configuration tuple, e.g. (('A', 'B', 'C'), 'A')
- **features** (*dict*) – Dictionary of {str: list} of generic features for a model, not considering the configuration. For example: {'CPM_FORM': [sympy.S.One, v.T, v.T**2, v.T**3]}

Returns Dictionary of {feature: [candidate_models])

Return type dict

Notes

Currently only works for binary and ternary interactions.

Candidate models match the following spec: 1. Candidates with multiple features specified will have 2. orders of parameters (L0, L0 and L1, ...) have the same number of temperatures

Note that high orders of parameters with multiple temperatures are not required to contain all the temperatures of the low order parameters. For example, the following parameters can be generated L0: A L1: A + BT

espei.parameter_selection.model_building.**build_feature_sets**(*temperature_features*, *interaction_features*)
Return a list of broadcasted features

Parameters

- **temperature_features** (*list*) – List of temperature features that will become a successive_list, such as [TlogT, T-1, T2]
- **interaction_features** (*list*) – List of interaction features that will become a successive_list, such as [YS, YS*Z, YS*Z**2]

Returns

Return type list

Notes

This allows two sets of features, e.g. [TlogT, T-1, T2] and [YS, YS*Z, YS*Z**2] and generates a list of feature sets where the temperatures and interactions are broadcasted successively.

Generates candidate feature sets like: L0: A + BT, L1: A L0: A , L1: A + BT

but **not** lists that are not successive: L0: A + BT, L1: Nothing, L2: A L0: Nothing, L1: A + BT

There's still some debate whether it makes sense from an information theory perspective to add a L1 B term without an L0 B term. However this might be more representative of how people usually model thermodynamics.

Does not distribute multiplication/sums or make assumptions about the elements of the feature lists. They can be strings, ints, objects, tuples, etc..

The number of features (related to the complexity) is a geometric series. For N temperature features and M interaction features, the total number of feature sets should be $N*(1-N**M)/(1-N)$. If $N=1$, then there are M total feature sets.

espei.parameter_selection.model_building.**make_successive**(*xs*)

Return a list of successive combinations

Parameters **xs** (*list*) – List of elements, e.g. [X, Y, Z]

Returns List of combinations where each combination include all the preceding elements

Return type list

Examples

```
>>> make_successive(['W', 'X', 'Y', 'Z'])
[['W'], ['W', 'X'], ['W', 'X', 'Y'], ['W', 'X', 'Y', 'Z']]
```

espei.parameter_selection.redlich_kister module

Tools for construction Redlich-Kister polynomials used in parameter selection.

espei.parameter_selection.redlich_kister.**calc_interaction_product**(*site_fractions*)

Calculate the interaction product for sublattice configurations

Parameters **site_fractions** (*list*) – List of sublattice configurations. *Sites on each sublattice be in order with respect to the elements in the sublattice.* The list should be 3d of (configurations, sublattices, values)

Returns List of interaction products, Z, for each sublattice

Return type list

Examples

```
>>> # interaction product for an (A) configuration
>>> calc_interaction_product([[1.0]])  # doctest: +ELLIPSIS
[1.0]
>>> # interaction product for [(A,B), (A,B)(A)] configurations that are equal
>>> calc_interaction_product([[[0.5, 0.5]], [[0.5, 0.5], 1]])  # doctest:␣
↪+ELLIPSIS
[0.0, 0.0]
>>> # interaction product for an [(A,B)] configuration
>>> calc_interaction_product([[[0.1, 0.9]]])  # doctest: +ELLIPSIS
[-0.8]
>>> # interaction product for an [(A,B)(A,B)] configuration
>>> calc_interaction_product([[[0.2, 0.8], [0.4, 0.6]]])  # doctest: +ELLIPSIS
[0.12]
>>> # ternary case, (A,B,C) interaction
>>> calc_interaction_product([[[0.333, 0.333, 0.334]]])
[[0.333, 0.333, 0.334]]
>>> # ternary 2SL case, (A,B,C)(A) interaction
>>> calc_interaction_product([[[0.333, 0.333, 0.334], 1.0]])
[[0.333, 0.333, 0.334]]
```

espei.parameter_selection.redlich_kister.**calc_site_fraction_product**(*site_fractions*)

Calculate the site fraction product for sublattice configurations

Parameters **site_fractions** (*list*) – List of sublattice configurations. The list should be 3d of (configurations, sublattices, values)

Returns List of site fraction products, YS, for each sublattice

Return type list

Examples

```
>>> # site fraction product for an (A,B)(A) configuration
>>> calc_site_fraction_product([[[0.2, 0.8], 1.0]])  # doctest: +ELLIPSIS
[0.16...]
>>> # site fraction product for [(A,B)(A), (A,B)(A)] configurations
>>> calc_site_fraction_product([[[0.2, 0.8], 1.0], [[0.3, 0.7], 1.0]])  #␣
↪doctest: +ELLIPSIS
[0.16..., 0.21]
>>> # site fraction product for [(A,B)(A,B)] configurations
>>> calc_site_fraction_product([[[0.2, 0.8], [0.4, 0.6]]])  # doctest: +ELLIPSIS
[0.0384...]
>>> # ternary case, (A,B,C) interaction
>>> calc_site_fraction_product([[[0.25, 0.25, 0.5]]])
[0.03125]
```

espei.parameter_selection.selection module

Fit, score and select models

espei.parameter_selection.selection.**fit_model**(*feature_matrix*, *data_quantities*, *ridge_alpha*, *weights=None*)

Return model coefficients fit by scikit-learn's LinearRegression

Parameters

- **feature_matrix** (*ndarray*) – (M*N) regressor matrix. The transformed model inputs (y_i, T, P, etc.)
- **data_quantities** (*ndarray*) – (M,) response vector. Target values of the output (e.g. HM_MIX) to reproduce.
- **ridge_alpha** (*float*) – Value of the $alpha$ hyperparameter used in ridge regression. Defaults to 1.0e-100, which should be degenerate with ordinary least squares regression. For now, the parameter is applied to all features.

Returns List of model coefficients of shape (N,)

Return type list

Notes

Solve Ax = b. *x* are the desired model coefficients. *A* is the 'feature_matrix'. *b* corrresponds to 'data_quantities'.

espei.parameter_selection.selection.**score_model**(*feature_matrix*, *data_quantities*, *model_coefficients*, *feature_list*, *weights*, *aicc_factor=None*, *rss_numerical_limit=1e-16*)

Use the AICc to score a model that has been fit.

Parameters

- **feature_matrix** (*ndarray*) – (M*N) regressor matrix. The transformed model inputs (y_i, T, P, etc.)
- **data_quantities** (*ndarray*) – (M,) response vector. Target values of the output (e.g. HM_MIX) to reproduce.
- **model_coefficients** (*list*) – List of fitted model coefficients to be scored. Has shape (N,).
- **feature_list** (*list*) – Polynomial coefficients corresponding to each column of 'feature_matrix'. Has shape (N,). Purely a logging aid.
- **aicc_factor** (*float*) – Multiplication factor for the AICc's parameter penalty.
- **rss_numerical_limit** (*float*) – Anything with an absolute value smaller than this is set to zero.

Returns A model score

Return type float

Notes

Solve Ax = b, where 'feature_matrix' is A and 'data_quantities' is b.

The likelihood function is a simple least squares with no regularization. The form of the AIC is valid under assumption all sample variances are random and Gaussian, model is univariate. It is assumed the model here is univariate with T.

espei.parameter_selection.selection.**select_model**(*candidate_models*, *ridge_alpha*, *weights*, *aicc_factor=None*)

Select a model from a series of candidates by fitting and scoring them

Parameters

- **candidate_models** (*list*) – List of tuples of (features, feature_matrix, data_quantities)
- **ridge_alpha** (*float*) – Value of the $alpha$ hyperparameter used in ridge regression. Defaults to 1.0e-100, which should be degenerate with ordinary least squares regression. For now, the parameter is applied to all features.
- **aicc_factor** (*float*) – Multiplication factor for the AICc's parameter penalty.

Returns Tuple of (feature_list, model_coefficients) for the highest scoring model

Return type tuple

espei.parameter_selection.ternary_parameters module

Build fittable models for ternary parameter selection

espei.parameter_selection.ternary_parameters.**build_ternary_feature_matrix**(*prop*, *candidate_models*, *desired_data*)

Return an MxN matrix of M data sample and N features.

Parameters

- **prop** (*str*) – String name of the property, e.g. 'HM_MIX'
- **candidate_models** (*list*) – List of SymPy parameters that can be fit for this property.
- **desired_data** (*dict*) – Full dataset dictionary containing values, conditions, etc.

Returns An MxN matrix of M samples (from desired data) and N features.

Return type numpy.ndarray

espei.parameter_selection.utils module

Tools used across parameter selection modules

espei.parameter_selection.utils.**get_data_quantities**(*desired_property*, *fixed_model*, *fixed_portions*, *data*)

Parameters

- **desired_property** (*str*) – String property corresponding to the features that could be fit, e.g. HM, SM_FORM, CPM_MIX
- **fixed_model** (*pycalphad.Model*) – Model with all lower order (in composition) terms already fit. Pure element reference state (GHSER functions) should be set to zero.
- **fixed_portions** (*List[sympy.Expr]*) – SymPy expressions for model parameters and interaction productions for higher order (in T) terms for this property, e.g. [0, 3.0*YS*v.T]. In [qty]/mole-formula.
- **data** (*List[Dict[str, Any]]*) – ESPEI single phase datasets for this property.

Returns **np.ndarray[** – Ravelled data quantities in [qty]/mole-formula

Return type

]

Notes

pycalphad Model parameters (and therefore fixed_portions) are stored as per mole-formula quantites, but the calculated properties and our data are all in [qty]/mole-atoms. We multiply by mole-atoms/mole-formula to convert the units to [qty]/mole-formula.

espei.parameter_selection.utils.**shift_reference_state**(*desired_data*, *feature_transform*, *fixed_model*, *mole_atoms_per_mole_formula_unit*)

Shift _MIX or _FORM data to a common reference state in per mole-atom units.

Parameters

- **desired_data** (*List[Dict[str, Any]]*) – ESPEI single phase dataset
- **feature_transform** (*Callable*) – Function to transform an AST for the GM property to the property of interest, i.e. entropy would be `lambda GM: -sympy.diff(GM, v.T)`
- **fixed_model** (*pycalphad.Model*) – Model with all lower order (in composition) terms already fit. Pure element reference state (GHSER functions) should be set to zero.
- **mole_atoms_per_mole_formula_unit** (*float*) – Number of moles of atoms in every mole atom unit.

Returns Data for this feature in [qty]/mole-formula in a common reference state.

Return type np.ndarray

Raises `ValueError`

Notes

pycalphad Model parameters are stored as per mole-formula quantites, but the calculated properties and our data are all in [qty]/mole-atoms. We multiply by mole-atoms/mole-formula to convert the units to [qty]/mole-formula.

Module contents

Submodules

espei.analysis module

Tools for analyzing ESPEI runs

espei.analysis.**truncate_arrays**(*trace_array*, *prob_array=None*)

Return slides of ESPEI output arrays with any empty remaining iterations (zeros) removed.

Parameters

- **trace_array** (*np.ndarray*) – Array of the trace from an ESPEI run. Should have shape (chains, iterations, parameters)
- **prob_array** (*np.ndarray*) – Array of the lnprob output from an ESPEI run. Should have shape (chains, iterations)

Returns A slide of the zeros-removed trace array is returned if only the trace is passed. Otherwise a tuple of both the trace and lnprob are returned.

Return type np.ndarry or (np.ndarray, np.ndarray)

espei.citing module

Define citations for ESPEI

espei.core_utils module

Internal utilities for developer use. May not be useful to users.

espei.core_utils.**get_data**(*comps*, *phase_name*, *configuration*, *symmetry*, *datasets*, *prop*)
Return list of cleaned single phase datasets matching the passed arguments.

Parameters

- **comps** (*list*) – List of string component names
- **phase_name** (*str*) – Name of phase
- **configuration** (*tuple*) – Sublattice configuration as a tuple, e.g. ("CU", ("CU", "MG"))
- **symmetry** (*list of lists*) – List of sublattice indices with symmetry
- **datasets** (*espei.utils.PickleableTinyDB*) – Database of datasets to search for data
- **prop** (*list*) – String name of the property of interest.

Returns List of datasets matching the arguments.

Return type list

espei.core_utils.**get_prop_data**(*comps*, *phase_name*, *prop*, *datasets*, *additional_query=None*)
Return datasets that match the components, phase and property

Parameters

- **comps** (*list*) – List of components to get data for
- **phase_name** (*str*) – Name of the phase to get data for
- **prop** (*str*) – Property to get data for
- **datasets** (*espei.utils.PickleableTinyDB*) – Datasets to search for data
- **additional_query** (*tinydb.Query*) – A TinyDB Query object to search for. If None, a Query() will be created that does nothing.

Returns List of dictionary datasets that match the criteria

Return type list

espei.core_utils.**get_samples**(*desired_data*)
Return the data values from desired_data, transformed to interaction products.

Parameters **desired_data** (*list*) – List of matched desired data, e.g. for a single property

Returns Tuples of (temperature, (site fraction product, interaction product))

Return type List[Tuple[float, Tuple[float, float]]]

Notes

Transforms data to interaction products, e.g. YS*{}^{xs}G=YS*XS*DXS^{n} {}^{n}L

espei.core_utils.**get_weights**(*desired_data*)

espei.core_utils.**ravel_conditions**(*values*, **conditions*, ***kwargs*)

Broadcast and flatten conditions to the shape dictated by the values.

Special handling for ZPF data that does not have nice array values.

Parameters

- **values** (*list*) – Multidimensional lists of values
- **conditions** (*list*) – List of conditions to broadcast. Must be the same length as the number of dimensions of the values array. In code, the following must be True: *all([s == len(cond) for s, cond in zip(values.shape, conditions)])*
- **zpf** (*bool, optional*) – Whether to consider values as a special case of ZPF data (not an even grid of conditions) Default is False

Returns Tuple of ravelled conditions

Return type tuple

Notes

The current implementation of ZPF data only has the shape for one condition and this assumption is hardcoded in various places.

Here we try to be as general as possible by explicitly calculating the shape of the ZPF values.

A complication of this is that the user of this function must pass the correct conditions because usually T and P are specified in ZPF (but, again, only one can actually be a condition given the current shape).

espei.core_utils.**ravel_zpf_values**(*desired_data*, *independent_comps*, *conditions=None*)

Unpack the phases and compositions from ZPF data. Dependent components are converted to independent components.

Parameters

- **desired_data** (*espei.utils.PickleableTinyDB*) – The selected data
- **independent_comps** (*list*) – List of indepdendent components. Used for mass balance component conversion
- **conditions** (*dict*) – Conditions to filter for. Right now only considers fixed temperatures

Returns A dictonary of list of lists of tuples. Each dictionary key is the number of phases in equilibrium, e.g. a key "2" might have values [[(PHASE_NAME_1, {'C1': X1, 'C2': X2}, refkey), (PHASE_NAME_2, {'C1': X1, 'C2': X2}, refkey)], ...] Three would have three inner tuples and so on.

Return type dict

espei.core_utils.**recursive_map**(*f*, *x*)

map, but over nested lists

Parameters

- **f** (*callable*) – Function to apply to x

- **x** (*list or value*) – Value passed to v

Returns

Return type list or value

espei.core_utils.**symmetry_filter**(*x*, *config*, *symmetry*)
Return True if the candidate sublattice configuration has any symmetry which matches the phase model symmetry.

Parameters

- **x** (*dict*) – the candidate dataset 'solver' dict. Must contain the "sublattice_configurations" key
- **config** (*list*) – the configuration of interest: e.g. ['AL', ['AL', 'NI'], 'VA']
- **symmetry** (*list*) – tuple of tuples where each inner tuple is a group of equivalent sublattices. A value of ((0, 1), (2, 3, 4)) means that sublattices at indices 0 and 1 are symmetrically equivalent to each other and sublattices at indices 2, 3, and 4 are symetrically equivalent to each other.

Returns

Return type bool

espei.datasets module

exception espei.datasets.**DatasetError**
Bases: Exception

Exception raised when datasets are invalid.

espei.datasets.**add_ideal_exclusions**(*datasets*)
If there are single phase datasets present and none of them have an *excluded_model_contributions* key, add ideal exclusions automatically and emit a DeprecationWarning that this feature will be going away.

Parameters **datasets** (*PickleableTinyDB*) –

Returns

Return type PickleableTinyDB

espei.datasets.**apply_tags**(*datasets*, *tags*)
Modify datasets using the tags system

Parameters

- **datasets** (*PickleableTinyDB*) – Datasets to modify
- **tags** (*dict*) – Dictionary of {tag: update_dict}

Returns

Return type PickleableTinyDB

Notes

In general, everything replaces or is additive. We use the following update rules: 1. If the update value is a list, extend the existing list (empty list if key does not exist) 2. If the update value is scalar, override the previous (deleting any old value, if present) 3. If the update value is a dict, update the exist dict (empty dict if dict does not exist) 4. Otherwise, the value is updated, overriding the previous

Examples

```
>>> from espei.utils import PickleableTinyDB
>>> from tinydb.storages import MemoryStorage
>>> ds = PickleableTinyDB(storage=MemoryStorage)
>>> doc_id = ds.insert({'tags': ['dft'], 'excluded_model_contributions': ['contrib
↪']})
>>> my_tags = {'dft': {'excluded_model_contributions': ['idmix', 'mag'], 'weight
↪': 5.0}}
>>> from espei.datasets import apply_tags
>>> apply_tags(ds, my_tags)
>>> all_data = ds.all()
>>> all(d['excluded_model_contributions'] == ['contrib', 'idmix', 'mag'] for d in␣
↪all_data)
True
>>> all(d['weight'] == 5.0 for d in all_data)
True
```

espei.datasets.**check_dataset**(*dataset*)
Ensure that the dataset is valid and consistent.

Currently supports the following validation checks: * data shape is valid * phases and components used match phases and components entered * individual shapes of keys, such as ZPF, sublattice configs and site ratios

Planned validation checks: * all required keys are present

Note that this follows some of the implicit assumptions in ESPEI at the time of writing, such that conditions are only P, T, configs for single phase and essentially only T for ZPF data.

Parameters **dataset** (*dict*) – Dictionary of the standard ESPEI dataset.

Returns

Return type None

Raises *DatasetError* – If an error is found in the dataset

espei.datasets.**clean_dataset**(*dataset*)
Clean an ESPEI dataset dictionary.

Parameters **dataset** (*dict*) – Dictionary of the standard ESPEI dataset. dataset : dic

Returns Modified dataset that has been cleaned

Return type dict

Notes

Assumes a valid, checked dataset. Currently handles * Converting expected numeric values to floats

espei.datasets.**load_datasets**(*dataset_filenames*)
Create a PickelableTinyDB with the data from a list of filenames.

Parameters **dataset_filenames** (*[str]*) – List of filenames to load as datasets

Returns

Return type PickleableTinyDB

espei.datasets.**recursive_glob**(*start*, *pattern='*.json'*)
Recursively glob for the given pattern from the start directory.

Parameters

- **start** (*str*) – Path of the directory to walk while for file globbing
- **pattern** (*str*) – Filename pattern to match in the glob.

Returns List of matched filenames

Return type [str]

espei.espei_script module

Automated fitting script.

A minimal run must specify an input.json and a datasets folder containing input files.

espei.espei_script.**get_dask_config_paths**()

espei.espei_script.**get_run_settings**(*input_dict*)
Validate settings from a dict of possible input.

Performs the following actions: 1. Normalize (apply defaults) 2. Validate against the schema

Parameters **input_dict** (*dict*) – Dictionary of input settings

Returns Validated run settings

Return type dict

Raises ValueError

espei.espei_script.**log_version_info**()
Print version info to the log

espei.espei_script.**main**()
Handle starting ESPEI from the command line. Parse command line arguments and input file.

espei.espei_script.**run_espei**(*run_settings*)
Wrapper around the ESPEI fitting procedure, taking only a settings dictionary.

Parameters **run_settings** (*dict*) – Dictionary of input settings

Returns

Return type Either a Database (for generate parameters only) or a tuple of (Database, sampler)

espei.mcmc module

Legacy module for running MCMC in ESPEI

espei.mcmc.**mcmc_fit**(*dbf*, *datasets*, *iterations=1000*, *save_interval=1*, *chains_per_parameter=2*, *chain_std_deviation=0.1*, *scheduler=None*, *tracefile=None*, *probfile=None*, *restart_trace=None*, *deterministic=True*, *prior=None*, *mcmc_data_weights=None*)
Run MCMC via the EmceeOptimizer class

Parameters

- **dbf** (*Database*) – A pycalphad Database to fit with symbols to fit prefixed with *VV* followed by a number, e.g. *VV0001*
- **datasets** (*PickleableTinyDB*) – A database of single- and multi-phase data to fit
- **iterations** (*int*) – Number of trace iterations to calculate in MCMC. Default is 1000 iterations.

- **save_interval** (*int*) – interval of iterations to save the tracefile and probfile
- **chains_per_parameter** (*int*) – number of chains for each parameter. Must be an even integer greater or equal to 2. Defaults to 2.
- **chain_std_deviation** (*float*) – standard deviation of normal for parameter initialization as a fraction of each parameter. Must be greater than 0. Default is 0.1, which is 10%.
- **scheduler** (*callable*) – Scheduler to use with emcee. Must implement a map method.
- **tracefile** (*str*) – filename to store the trace with NumPy.save. Array has shape (chains, iterations, parameters)
- **probfile** (*str*) – filename to store the log probability with NumPy.save. Has shape (chains, iterations)
- **restart_trace** (*np.ndarray*) – ndarray of the previous trace. Should have shape (chains, iterations, parameters)
- **deterministic** (*bool*) – If True, the emcee sampler will be seeded to give deterministic sampling draws. This will ensure that the runs with the exact same database, chains_per_parameter, and chain_std_deviation (or restart_trace) will produce exactly the same results.
- **prior** (*str*) – Prior to use to generate priors. Defaults to 'zero', which keeps backwards compatibility. Can currently choose 'normal', 'uniform', 'triangular', or 'zero'.
- **mcmc_data_weights** (*dict*) – Dictionary of weights for each data type, e.g. {'ZPF': 20, 'HM': 2}

espei.paramselect module

The paramselect module handles automated parameter selection for linear models.

Automated Parameter Selection End-members

Note: All magnetic parameters from literature for now. Note: No fitting below 298 K (so neglect third law issues for now).

For each step, add one parameter at a time and compute AICc with max likelihood.

Cp - TlnT, T**2, T**-1, T**3 - 4 candidate models (S and H only have one required parameter each. Will fit in full MCMC procedure)

Choose parameter set with best AICc score.

espei.paramselect.**fit_formation_energy**(*dbf*, *comps*, *phase_name*, *configuration*, *symmetry*, *datasets*, *ridge_alpha=None*, *aicc_phase_penalty=None*, *features=None*)

Find suitable linear model parameters for the given phase. We do this by successively fitting heat capacities, entropies and enthalpies of formation, and selecting against criteria to prevent overfitting. The "best" set of parameters minimizes the error without overfitting.

Parameters

- **dbf** (*Database*) – pycalphad Database. Partially complete, so we know what degrees of freedom to fix.
- **comps** (*[str]*) – Names of the relevant components.
- **phase_name** (*str*) – Name of the desired phase for which the parameters will be found.

- **configuration** (*ndarray*) – Configuration of the sublattices for the fitting procedure.
- **symmetry** (*[[int]]*) – Symmetry of the sublattice configuration.
- **datasets** (*PickleableTinyDB*) – All the datasets desired to fit to.
- **ridge_alpha** (*float*) – Value of the $alpha$ hyperparameter used in ridge regression. Defaults to 1.0e-100, which should be degenerate with ordinary least squares regression. For now, the parameter is applied to all features.
- **aicc_feature_factors** (*dict*) – Map of phase name to feature to a multiplication factor for the AICc's parameter penalty.
- **features** (*dict*) – Maps "property" to a list of features for the linear model. These will be transformed from "GM" coefficients e.g., {"CPM_FORM": (v.T*sympy.log(v.T), v.T**2, v.T**-1, v.T**3)} (Default value = None)

Returns {feature: estimated_value}

Return type dict

espei.paramselect.**fit_ternary_interactions**(*dbf*, *phase_name*, *symmetry*, *endmembers*, *datasets*, *ridge_alpha=None*, *aicc_phase_penalty=None*)

Fit ternary interactions for a database in place

Parameters

- **dbf** (*Database*) – pycalphad Database to add parameters to
- **phase_name** (*str*) – Name of the phase to fit
- **symmetry** (*list*) – List of symmetric sublattices, e.g. [[0, 1, 2], [3, 4]]
- **endmembers** (*list*) – List of endmember tuples, e.g. [('CU', 'MG')]
- **datasets** (*PickleableTinyDB*) – TinyDB database of datasets
- **ridge_alpha** (*float*) – Value of the $alpha$ hyperparameter used in ridge regression. Defaults to 1.0e-100, which should be degenerate with ordinary least squares regression. For now, the parameter is applied to all features.

Returns Modified the Database in place

Return type None

espei.paramselect.**generate_parameters**(*phase_models*, *datasets*, *ref_state*, *excess_model*, *ridge_alpha=None*, *aicc_penalty_factor=None*, *dbf=None*)

Generate parameters from given phase models and datasets

Parameters

- **phase_models** (*dict*) – Dictionary of components and phases to fit.
- **datasets** (*PickleableTinyDB*) – database of single- and multi-phase to fit.
- **ref_state** (*str*) – String of the reference data to use, e.g. 'SGTE91' or 'SR2016'
- **excess_model** (*str*) – String of the type of excess model to fit to, e.g. 'linear'
- **ridge_alpha** (*float*) – Value of the $alpha$ hyperparameter used in ridge regression. Defaults to None, which falls back to ordinary least squares regression. For now, the parameter is applied to all features.
- **aicc_penalty_factor** (*dict*) – Map of phase name to feature to a multiplication factor for the AICc's parameter penalty.

- **dbf** (*Database*) – Initial pycalphad Database that can have parameters that would not be fit by ESPEI

Returns

Return type pycalphad.Database

espei.paramselect.**get_next_symbol**(*dbf*)
Return a string name of the next free symbol to set

Parameters **dbf** (*Database*) – pycalphad Database. Must have the `varcounter` attribute set to an integer.

Returns

Return type str

espei.paramselect.**phase_fit**(*dbf*, *phase_name*, *symmetry*, *subl_model*, *site_ratios*, *datasets*, *refdata*, *ridge_alpha*, *aicc_penalty=None*, *aliases=None*)
Generate an initial CALPHAD model for a given phase and sublattice model.

Parameters

- **dbf** (*Database*) – pycalphad Database to add parameters to.
- **phase_name** (*str*) – Name of the phase.
- **symmetry** (*[[int]]*) – Sublattice model symmetry.
- **subl_model** (*[[str]]*) – Sublattice model for the phase of interest.
- **site_ratios** (*[float]*) – Number of sites in each sublattice, normalized to one atom.
- **datasets** (*PickleableTinyDB*) – All datasets to consider for the calculation.
- **refdata** (*dict*) – Maps tuple(element, phase_name) -> SymPy object defining energy relative to SER
- **ridge_alpha** (*float*) – Value of the $alpha$ hyperparameter used in ridge regression. Defaults to 1.0e-100, which should be degenerate with ordinary least squares regression. For now, the parameter is applied to all features.
- **aicc_penalty** (*dict*) – Map of phase name to feature to a multiplication factor for the AICc's parameter penalty.
- **aliases** (*[str]*) – Alternative phase names. Useful for matching against reference data or other datasets. (Default value = None)

Returns Modifies the dbf.

Return type None

espei.plot module

Plotting of input data and calculated database quantities

espei.plot.**dataplot**(*comps*, *phases*, *conds*, *datasets*, *ax=None*, *plot_kwargs=None*, *tieline_plot_kwargs=None*)
Plot datapoints corresponding to the components, phases, and conditions.

Parameters

- **comps** (*list*) – Names of components to consider in the calculation.
- **phases** (*[]*) – Names of phases to consider in the calculation.

- **conds** (*dict*) – Maps StateVariables to values and/or iterables of values.
- **datasets** (*PickleableTinyDB*) –
- **ax** (*matplotlib.Axes*) – Default axes used if not specified.
- **plot_kwargs** (*dict*) – Additional keyword arguments to pass to the matplotlib plot function for points
- **tieline_plot_kwargs** (*dict*) – Additional keyword arguments to pass to the matplotlib plot function for tielines

Returns A plot of phase equilibria points as a figure

Return type matplotlib.Axes

Examples

```
>>> from espei.datasets import load_datasets, recursive_glob  # doctest: +SKIP
>>> from espei.plot import dataplot  # doctest: +SKIP
>>> datasets = load_datasets(recursive_glob('.', '*.json'))  # doctest: +SKIP
>>> my_phases = ['BCC_A2', 'CUMG2', 'FCC_A1', 'LAVES_C15', 'LIQUID']  # doctest:␣
↪+SKIP
>>> my_components = ['CU', 'MG' 'VA']  # doctest: +SKIP
>>> conditions = {v.P: 101325, v.T: (500, 1000, 10), v.X('MG'): (0, 1, 0.01)}  #␣
↪doctest: +SKIP
>>> dataplot(my_components, my_phases, conditions, datasets)  # doctest: +SKIP
```

espei.plot.**eqdataplot**(*eq*, *datasets*, *ax=None*, *plot_kwargs=None*)

Plot datapoints corresponding to the components and phases in the eq Dataset. A convenience function for dataplot.

Parameters

- **eq** (*xarray.Dataset*) – Result of equilibrium calculation.
- **datasets** (*PickleableTinyDB*) – Database of phase equilibria datasets
- **ax** (*matplotlib.Axes*) – Default axes used if not specified.
- **plot_kwargs** (*dict*) – Keyword arguments to pass to dataplot

Returns

Return type A plot of phase equilibria points as a figure

Examples

```
>>> from pycalphad import equilibrium, Database, variables as v  # doctest: +SKIP
>>> from pycalphad.plot.eqplot import eqplot  # doctest: +SKIP
>>> from espei.datasets import load_datasets, recursive_glob  # doctest: +SKIP
>>> datasets = load_datasets(recursive_glob('.', '*.json'))  # doctest: +SKIP
>>> dbf = Database('my_databases.tdb')  # doctest: +SKIP
>>> my_phases = list(dbf.phases.keys())  # doctest: +SKIP
>>> eq = equilibrium(dbf, ['CU', 'MG', 'VA'], my_phases, {v.P: 101325, v.T: (500,␣
↪1000, 10), v.X('MG'): (0, 1, 0.01)})  # doctest: +SKIP
>>> ax = eqplot(eq)  # doctest: +SKIP
>>> ax = eqdataplot(eq, datasets, ax=ax)  # doctest: +SKIP
```

espei.plot.**multiplot**(*dbf*, *comps*, *phases*, *conds*, *datasets*, *eq_kwargs=None*, *plot_kwargs=None*, *data_kwargs=None*)

Plot a phase diagram with datapoints described by datasets. This is a wrapper around pycalphad.equilibrium, pycalphad's eqplot, and dataplot.

Parameters

- **dbf** (*Database*) – pycalphad thermodynamic database containing the relevant parameters.
- **comps** (*list*) – Names of components to consider in the calculation.
- **phases** (*list*) – Names of phases to consider in the calculation.
- **conds** (*dict*) – Maps StateVariables to values and/or iterables of values.
- **datasets** (*PickleableTinyDB*) – Database of phase equilibria datasets
- **eq_kwargs** (*dict*) – Keyword arguments passed to pycalphad equilibrium()
- **plot_kwargs** (*dict*) – Keyword arguments passed to pycalphad eqplot()
- **data_kwargs** (*dict*) – Keyword arguments passed to dataplot()

Returns

Return type A phase diagram with phase equilibria data as a figure

Examples

```
>>> from pycalphad import Database, variables as v  # doctest: +SKIP
>>> from pycalphad.plot.eqplot import eqplot  # doctest: +SKIP
>>> from espei.datasets import load_datasets, recursive_glob  # doctest: +SKIP
>>> datasets = load_datasets(recursive_glob('.', '*.json'))  # doctest: +SKIP
>>> dbf = Database('my_databases.tdb')  # doctest: +SKIP
>>> my_phases = list(dbf.phases.keys())  # doctest: +SKIP
>>> multiplot(dbf, ['CU', 'MG', 'VA'], my_phases, {v.P: 101325, v.T: 1000, v.X('MG
↪'): (0, 1, 0.01)}, datasets)  # doctest: +SKIP
```

espei.plot.**plot_parameters**(*dbf*, *comps*, *phase_name*, *configuration*, *symmetry*, *datasets=None*, *fig=None*, *require_data=True*)

Plot parameters of interest compared with data in subplots of a single figure

Parameters

- **dbf** (*Database*) – pycalphad thermodynamic database containing the relevant parameters.
- **comps** (*list*) – Names of components to consider in the calculation.
- **phase_name** (*str*) – Name of the considered phase phase
- **configuration** (*tuple*) – Sublattice configuration to plot, such as ('CU', 'CU') or (('CU', 'MG'), 'CU')
- **symmetry** (*list*) – List of lists containing indices of symmetric sublattices e.g. [[0, 1], [2, 3]]
- **datasets** (*PickleableTinyDB*) – ESPEI datasets to compare against. If None, nothing is plotted.
- **fig** (*matplotlib.Figure*) – Figure to create with axes as subplots.

- **require_data** (*bool*) – If True, plot parameters that have data corresponding data. Defaults to True. Will raise an error for non-interaction configurations.

Returns

Return type None

Examples

```
>>> # plot the LAVES_C15 (Cu)(Mg) endmember
>>> plot_parameters(dbf, ['CU', 'MG'], 'LAVES_C15', ('CU', 'MG'), symmetry=None,␣
↪datasets=datasets)  # doctest: +SKIP
>>> # plot the mixing interaction in the first sublattice
>>> plot_parameters(dbf, ['CU', 'MG'], 'LAVES_C15', (('CU', 'MG'), 'MG'),␣
↪symmetry=None, datasets=datasets)  # doctest: +SKIP
```

espei.priors module

Classes and functions for retrieving statistical priors for given parameters.

class espei.priors.**DistributionParameter**(*parameter*, *param_type='absolute'*)

Bases: object

Handle generating absolute, scaling, shifting parameters.

Examples

```
>>> dp = DistributionParameter(5.0, 'absolute')  # always get back 5
>>> dp.value(1.0) == 5.0
True
>>> dp = DistributionParameter(-2.0, 'relative')  # multiply by -2
>>> dp.value(2.0) == -4.0
True
>>> dp = DistributionParameter(-1.0, 'shift_absolute')  # subtract 1
>>> dp.value(2.0) == 1.0
True
>>> dp = DistributionParameter(-0.5, 'shift_relative')  # subtract 1/2 value
>>> dp.value(2.0) == 1.0
True
```

SUPPORTED_TYPES = ('absolute', 'relative', 'shift_absolute', 'shift_relative', 'identi

value(*p*)

Return the distribution parameter value modified by the parameter and type.

Parameters **p** (*float*) – Input parameter to modify.

Returns

Return type float

class espei.priors.**PriorSpec**(*name*, ***parameters*)

Bases: object

Specification template for instantiating priors.

SUPPORTED_PRIORS = ('normal', 'uniform', 'triangular', 'zero')

get_prior(*value*)
Instantiate a prior as described in the spec

Examples

```
>>> import numpy as np
>>> from espei.priors import PriorSpec
>>> tri_spec = {'name': 'triangular', 'loc_shift_relative': -0.5, 'scale_
↪shift_relative': 0.5, 'c': 0.5}
>>> np.isneginf(PriorSpec(**tri_spec).get_prior(10).logpdf(5.1))
False
>>> np.isneginf(PriorSpec(**tri_spec).get_prior(10).logpdf(4.9))
True
```

espei.priors.**build_prior_specs**(*prior_spec*, *parameters*)
Get priors from given parameters

Parameters

- **prior_spec** (*PriorSpec or dict*) – Either a prior spec dict (to instantiate), a PriorSpec, or a list of either. If a list is passed, it must correspond to the parameters.
- **parameters** (*list*) – List of parameters that the priors will be instantiated by

Returns

Return type [*PriorSpec*]

Examples

```
>>> s_norm = {'name': 'normal', 'scale_relative': 0.1, 'loc_identity': 1.0}
>>> len(build_prior_specs(s_norm, [10, 100])) == 2
True
>>> s_tri = {'name': 'triangular', 'loc_shift_relative': -0.5, 'scale_shift_
↪relative': 0.5, 'c': 0.5}
>>> from espei.priors import PriorSpec
>>> len(build_prior_specs([s_norm, PriorSpec(**s_tri)], [10, 100])) == 2
True
```

class espei.priors.**rv_zero**(**args*, ***kwargs*)
Bases: object

A simple class that mimics the scipy.stats.rv_continuous object's logpdf method, always returning zero.

This class mainly exists for backwards compatibility where no prior is specified.

Examples

```
>>> import numpy as np
>>> rv = rv_zero()
>>> np.isclose(rv.logpdf(-np.inf), 0.0)
True
>>> np.isclose(rv.logpdf(1.0), 0.0)
True
>>> np.isclose(rv.logpdf(0.0), 0.0)
True
```

logpdf(**args*, ***kwargs*)

espei.refdata module

Create a new *Mock* object. *Mock* takes several optional arguments that specify the behaviour of the Mock object:

- *spec*: This can be either a list of strings or an existing object (a class or instance) that acts as the specification for the mock object. If you pass in an object then a list of strings is formed by calling dir on the object (excluding unsupported magic attributes and methods). Accessing any attribute not in this list will raise an *AttributeError*.

  If *spec* is an object (rather than a list of strings) then *mock.__class__* returns the class of the spec object. This allows mocks to pass *isinstance* tests.

- *spec_set*: A stricter variant of *spec*. If used, attempting to *set* or get an attribute on the mock that isn't on the object passed as *spec_set* will raise an *AttributeError*.

- *side_effect*: A function to be called whenever the Mock is called. See the *side_effect* attribute. Useful for raising exceptions or dynamically changing return values. The function is called with the same arguments as the mock, and unless it returns *DEFAULT*, the return value of this function is used as the return value.

  If *side_effect* is an iterable then each call to the mock will return the next value from the iterable. If any of the members of the iterable are exceptions they will be raised instead of returned.

- *return_value*: The value returned when the mock is called. By default this is a new Mock (created on first access). See the *return_value* attribute.

- *wraps*: Item for the mock object to wrap. If *wraps* is not None then calling the Mock will pass the call through to the wrapped object (returning the real result). Attribute access on the mock will return a Mock object that wraps the corresponding attribute of the wrapped object (so attempting to access an attribute that doesn't exist will raise an *AttributeError*).

  If the mock has an explicit *return_value* set then calls are not passed to the wrapped object and the *return_value* is returned instead.

- *name*: If the mock has a name then it will be used in the repr of the mock. This can be useful for debugging. The name is propagated to child mocks.

Mocks can also be called with arbitrary keyword arguments. These will be used to set attributes on the mock after it is created.

espei.rstate module

espei.sublattice_tools module

Utilities for manipulating sublattice models.

espei.sublattice_tools.**canonical_sort_key**(*x*)
Wrap strings in tuples so they'll sort.

Parameters **x** (*list*) – List of strings to sort

Returns tuple of strings that can be sorted

Return type tuple

espei.sublattice_tools.**canonicalize**(*configuration*, *equivalent_sublattices*)
Sort a sequence with symmetry. This routine gives the sequence a deterministic ordering while respecting symmetry.

Parameters

- **configuration** (*[str]*) – Sublattice configuration to sort.
- **equivalent_sublattices** (*{{int}}*) – Indices of 'configuration' which should be equivalent by symmetry, i.e., [[0, 4], [1, 2, 3]] means permuting elements 0 and 4, or 1, 2 and 3, respectively, has no effect on the equivalence of the sequence.

Returns sorted tuple that has been canonicalized.

Return type str

espei.sublattice_tools.**endmembers_from_interaction**(*configuration*)
For a given configuration with possible interactions, return all the endmembers

espei.sublattice_tools.**generate_endmembers**(*sublattice_model*, *symmetry=None*)
Return all the unique endmembers by symmetry for a given sublattice model.

Parameters

- **sublattice_model** (*list of lists*) – General sublattice model, with each sublattice as a sublist.
- **symmetry** (*list of lists, optional*) – List of lists containing symmetrically equivalent sublattice indices. If None (default), all endmembers will be returned.

Returns List of endmember tuples

Return type list

Examples

```
>>> subl_model = [['A', 'B'], ['A','B']]
>>> generate_endmembers(subl_model)  # four endmembers
[('A', 'A'), ('A', 'B'), ('B', 'A'), ('B', 'B')]
>>> # three endmembers, ('A', 'B') is equivalent to ('B', 'A') by symmetry.
>>> generate_endmembers(subl_model, [[0, 1]])  # the first and second sublattices␣
↪are symmetrically equivalent.
[('A', 'A'), ('A', 'B'), ('B', 'B')]
```

espei.sublattice_tools.**generate_interactions**(*endmembers*, *order*, *symmetry*)
Returns a list of sorted interactions of a certain order

Parameters

- **endmembers** (*list*) – List of tuples/strings of all endmembers (including symmetrically equivalent)
- **order** (*int*) – Highest expected interaction order, e.g. ternary interactions should be 3
- **symmetry** (*list of lists*) – List of lists containing symmetrically equivalent sublattice indices, e.g. [[0, 1], [2, 3]] means that sublattices 0 and 1 are equivalent and sublattices 2 and 3 are also equivalent.

Returns List of interaction tuples, e.g. [('A', ('A', 'B'))]

Return type list

espei.sublattice_tools.**generate_symmetric_group**(*configuration*, *symmetry*)
For a particular configuration and list of sublattices with symmetry, generate all the symmetrically equivalent configurations.

Parameters

- **configuration** (*tuple*) – Tuple of a sublattice configuration.

- **symmetry** (*list of lists*) – List of lists containing symmetrically equivalent sublattice indices, e.g. [[0, 1], [2, 3]] means that sublattices 0 and 1 are equivalent and sublattices 2 and 3 are also equivalent.

Returns Tuple of configuration tuples that are all symmetrically equivalent.

Return type tuple

espei.sublattice_tools.**interaction_test**(*configuration*, *order=None*)

Returns True if the configuration has an interaction

Parameters **order** (*int, optional*) – Specific order to check for. E.g. a value of 3 checks for ternary interactions

Returns True if there is an interaction.

Return type bool

Examples

```
>>> configuration = [['A'], ['A','B']]
>>> interaction_test(configuration)
True
>>> interaction_test(configuration, order=2)
True
>>> interaction_test(configuration, order=3)
False
```

espei.sublattice_tools.**recursive_tuplify**(*x*)

Recursively convert a nested list to a tuple

espei.sublattice_tools.**sorted_interactions**(*interactions*, *max_interaction_order*, *symmetry*)

Return interactions sorted by interaction order

Parameters

- **interactions** (*list*) – List of tuples/strings of potential interactions
- **max_interaction_order** (*int*) – Highest expected interaction order, e.g. ternary interactions should be 3
- **symmetry** (*list of lists*) – List of lists containing symmetrically equivalent sublattice indices, e.g. [[0, 1], [2, 3]] means that sublattices 0 and 1 are equivalent and sublattices 2 and 3 are also equivalent.

Returns Sorted list of interactions

Return type list

Notes

Sort by number of full interactions, e.g. (A:A,B) is before (A,B:A,B) The goal is to return a sort key that can sort through multiple interaction orders, e.g. (A:A,B,C), which should be before (A,B:A,B,C), which should be before (A,B,C:A,B,C).

espei.sublattice_tools.**tuplify**(*x*)

Convert a list to a tuple, or wrap an object in a tuple if it's not a list or tuple.

espei.utils module

Utilities for ESPEI

Classes and functions defined here should have some reuse potential.

espei.utils.**add_bibtex_to_bib_database**(*bibtex*, *bib_db=None*)
Add entries from a BibTeX file to the bibliography database

Parameters

- **bibtex** (*str*) – Either a multiline string, a path, or a file-like object of a BibTeX file
- **bib_db** (*PickleableTinyDB*) – Database to put the BibTeX entries. Defaults to a module-level default database

Returns

Return type The modified bibliographic database

espei.utils.**bib_marker_map**(*bib_keys*, *markers=None*)
Return a dict with reference keys and marker dicts

Parameters

- **bib_keys** –
- **markers** (*list*) – List of 2-tuples of ('fillstyle', 'marker') e.g. [('top', 'o'), ('full', 's')]. Defaults to cycling through the filled markers, the different fill styles.

Returns Dictionary with bib_keys as keys, dict values of formatted strings and marker dicts

Return type dict

Examples

```
>>> mm = bib_marker_map(['otis2016', 'bocklund2018'])
>>> mm == {'bocklund2018': {'formatted': 'bocklund2018', 'markers': {'fillstyle':
↪'none', 'marker': 'o'}}, 'otis2016': {'formatted': 'otis2016', 'markers': {
↪'fillstyle': 'none', 'marker': 'v'}}}
True
```

espei.utils.**build_sitefractions**(*phase_name*, *sublattice_configurations*, *sublattice_occupancies*)
Convert nested lists of sublattice configurations and occupancies to a list of dictionaries. The dictionaries map SiteFraction symbols to occupancy values. Note that zero occupancy site fractions will need to be added separately since the total degrees of freedom aren't known in this function.

Parameters

- **phase_name** (*str*) – Name of the phase
- **sublattice_configurations** (*[[str]]*) – sublattice configuration
- **sublattice_occupancies** (*[[float]]*) – occupancy of each sublattice

Returns a list of site fractions over sublattices

Return type [[float]]

espei.utils.**database_symbols_to_fit**(*dbf*, *symbol_regex='^V[V]?([0-9]+)$'*)
Return names of the symbols to fit that match the regular expression

Parameters

- **dbf** (*Database*) – pycalphad Database
- **symbol_regex** (*str*) – Regular expression of the fitting symbols. Defaults to V or VV followed by one or more numbers.

Returns Context dictionary for different methods of calculation the error.

Return type dict

espei.utils.**flexible_open_string**(*obj*)

Return the string of a an object that is either file-like, a file path, or the raw string.

Parameters **obj** (*string-like or file-like*) – Either a multiline string, a path, or a file-like object

Returns

Return type str

espei.utils.**formatted_constituent_array**(*constituent_array*)

Given a constituent array of Species, return the classic CALPHAD-style interaction.

Parameters **constituent_array** (*list*) – List of sublattices, which are lists of Species in that sublattice

Returns String of the constituent array formatted in the classic CALPHAD style

Return type str

Examples

```
>>> from pycalphad import variables as v
>>> const_array = [[v.Species('CU'), v.Species('MG')], [v.Species('MG')]]
>>> formatted_constituent_array(const_array)
'CU,MG:MG'
```

espei.utils.**formatted_parameter**(*dbf*, *symbol*, *unique=True*)

Get the deconstructed pretty parts of the parameter/term a symbol belongs to in a Database.

Parameters

- **dbf** (*pycalphad.Database*) –
- **symbol** (*string or sympy.Symbol*) – Symbol in the Database to get the parameter for.
- **unique** (*bool*) – If True, will raise if more than one parameter containing the symbol is found.

Returns A named tuple with the following attributes: `phase_name`, `interaction`, `symbol`, `term`, `parameter_type` or `term_symbol` (which is just the Symbol * temperature term)

Return type FormattedParameter

espei.utils.**get_dask_config_paths**()

Return a list of configuration file paths for dask.

The last path in the list has the highest precedence.

Returns

Return type list

Examples

```
>>> config_files = get_dask_config_paths()
>>> len(config_files) > 1
True
```

espei.utils.**optimal_parameters**(*trace_array*, *lnprob_array*, *kth=0*)

Return the optimal parameters in the trace based on the highest likelihood. If kth is specified, return the kth set of *unique* optimal parameters.

Parameters

- **trace_array** (*ndarray*) – Array of shape (number of chains, iterations, number of parameters)
- **lnprob_array** (*ndarray*) – Array of shape (number of chains, iterations)
- **kth** (*int*) – Zero-indexed optimum. 0 (the default) is the most optimal solution. 1 is the second most optimal, etc.. Only *unique* solutions will be returned.

Returns

Return type Array of optimal parameters

Notes

It is ok if the calculation did not finish and the arrays are padded with zeros. The number of chains and iterations in the trace and lnprob arrays must match.

espei.utils.**parameter_term**(*expression*, *symbol*)

Determine the term, e.g. T*log(T) that belongs to the symbol in expression

Parameters

- **expression** –
- **symbol** –

espei.utils.**popget**(*d*, *key*, *default=None*)

Get the key from the dict, returning the default if not found.

Parameters

- **d** (*dict*) – Dictionary to get key from.
- **key** (*object*) – Key to get from the dictionary.
- **default** (*object*) – Default to return if key is not found in dictionary.

Returns

Return type object

Examples

```
>>> d = {'ABC': 5.0}
>>> popget(d, 'ZPF', 1.0) == 1.0
True
>>> popget(d, 'ABC', 1.0) == 5.0
True
```

espei.utils.**sigfigs**(*x*, *n*)
Round x to n significant digits

espei.utils.**unpack_piecewise**(*x*)

espei.validation module

Module contents

ESPEI

Part IV

Developer

CHAPTER 13

Contributing to ESPEI

13.1 Contributing to ESPEI

This is the place to start as a new ESPEI contributor.

The next sections lay out the basics of getting an ESPEI development set up and the development standards. Then the *Software design* sections walk through the key parts of the codebase.

13.1.1 Installing in develop mode

It is suggested to use ESPEI in development mode if you will be contributing features to the source code. As usual, you should install ESPEI into a virtual environment.

All of the dependencies can be installed either by conda or pip.

Then clone the source and install ESPEI in development mode with pip:

```
git clone https://github.com/PhasesResearchLab/espei.git
pip install --editable espei
```

Even if you use Anaconda, it is recommended that you use either `pip` or `python setup.py develop` to install ESPEI in development mode. This is because the `conda-build` tool, which would typically be used for this, is not well maintained at the time of writing.

Develop mode on Windows

Because of compiler issues, ESPEI's dependencies are challenging to install on Windows. As mentioned above, ideally the `conda-build` tool could be used, but it is not able to be used. Therefore the recommended way to install ESPEI is to

1. Install ESPEI into a virtual environment from Anaconda, pulling all of the packages with it
2. Remove ESPEI without removing the other packages

3. Install ESPEI in develop mode with pip or setuptools from the source repository

The steps to do this on the command line are as follows

```
conda create -n espei_dev espei
activate espei_dev
conda remove --force espei
git clone https://github.com/PhasesResearchLab/espei.git
pip install --editable espei
```

13.1.2 Tests

Even though much of ESPEI is devoted to being a multi-core, stochastic user tool, we strive to test all logic and functionality. We are continuously maintaining tests and writing tests for previously untested code.

As a general rule, any time you write a new function or modify an existing function you should write or maintain a test for that function.

ESPEI uses pytest as a test runner.

Some tips for testing:

- Ideally you would practicing test driven development by writing tests of your intended results before you write the function.
- If possible, keep the tests small and fast. If you do have a long running tests (longer than ~15 second run time) mark the test with the `@pytest.mark.slow` decorator.
- See the NumPy/SciPy testing guidelines for more tips

Running Tests

If you will be developing in ESPEI, it is likely that you'll want to run the test suite or build the documentation. The tests require the addition of the pytest, nose, and mock packages, while building the docs requires sphinx and sphinx_rtd_theme. These can be installed by running

```
conda install mock pytest nose sphinx sphinx_rtd_theme
```

The tests can be run from the root directory of the cloned repository:

```
pytest --doctest-modules tests espei
```

13.1.3 Style

Code style

For most naming and style, follow PEP8. One exception to PEP8 is regarding the line length, which we suggest a 120 character maximum, but may be longer within reason.

Code documentation

ESPEI uses the NumPy documentation style. All functions and classes should be documented with at least a description, parameters, and return values, if applicable.

Using `Examples` in the documentation is especially encouraged for utilities that are likely to be run by users. See *espei.plot.multiplot()* for an example.

If you add any new external (non-ESPEI) imports in any code, they must be added to the `MOCK_MODULES` list in `docs/conf.py`.

Web documention

Documentation on ESPEI is split into user tutorials, reference and developer documentation.

- Tutorials are resources for users new to ESPEI or new to certain features of ESPEI to be *guided* through typical actions.
- Reference pages should be concise articles that explain how to complete specific goals for users who know what they want to accomplish.
- Developer documentation should describe what should be considered when contributing source code back to ESPEI.

You can check changes you make to the documentation by going to the documentation folder in the root repository `cd docs/`. Running the command `make html && cd build/html && python3 -m http.server && cd ../.. && make clean` from that folder will build the docs and run them on a local HTTP server. You can see the documentation when the server is running by visting the URL at the end of the output, usually `localhost port 8000 <http://0.0.0.0:8000>``_. When you are finished, type ``Ctrl-C` to stop the server and the command will clean up the build for you.

Make sure to fix any warnings that come up if you are adding documentation.

Building Documentation

The docs can be built by running the docs/Makefile (or docs/make.bat on Windows). Then Python can be used to serve the html files in the _build directory and you can visit `http://localhost:8000` in your broswer to see the built documentation.

For Unix systems:

```
cd docs
make html
cd _build/html
python -m http.server
```

Windows:

```
cd docs
make.bat html
cd _build\html
python -m http.server
```

13.1.4 Logging

Since ESPEI is intended to be run by users, we must provide useful feedback on how their runs are progressing. ESPEI uses the logging module to allow control over verbosity of the output.

There are 5 different logging levels provided by Python. They should be used as follows:

Critical or Error (`logging.critical` or `logging.error`) Never use these. These log levels would only be used when there is an unrecoverable error that requires the run to be stopped. In that case, it is better to `raise` an appropriate error instead.

Warning (`logging.warning`) Warnings are best used when we are able to recover from something bad that has happened. The warning should inform the user about potentially incorrect results or let them know about something they have the potential to fix. Again, anything unrecoverable should not be logged and should instead be raised with a good error message.

Info (`logging.info`) Info logging should report on the progress of the program. Usually info should give feedback on milestones of a run or on actions that were taken as a result of a user setting. An example of a milestone is starting and finishing parameter generation. An example of an action taken as a result of a user setting is the logging of the number of chains in an mcmc run.

Debug (`logging.debug`) Debugging is the lowest level of logging we provide in ESPEI. Debug messages should consist of possibly useful information that is beyond the user's direct control. Examples are the values of initial parameters, progress of checking datasets and building phase models, and the acceptance ratios of MCMC iterations.

CHAPTER 14

Software design

14.1 Software design

The following sections elaborate on the design principles on the software side. The goal is to make it clear how different modules in ESPEI fit together and where to find specific functionality to override or improve.

ESPEI provides tools to

1. Parameterize CALPHAD models by optimizing the compromise between model accuracy and complexity. We typically call this parameter generation or model selection.
2. Fit parameterized CALPHAD models to multi-phase or other custom data with uncertainty quantification via Markov chain Monte Carlo

14.1.1 API

ESPEI has two levels of API that users should expect to interact with:

1. Input from YAML files on the command line (via `espei --input <input_file>` or by Python via the `espei.espei_script.run_espei` function
2. Work directly with the Python functions for parameter selection `espei.paramselect.generate_parameters` and MCMC `espei.mcmc.mcmc_fit`

YAML files are the recommended way to use ESPEI and should have a way to express most if not all of the options that the Python functions support. The schema for YAML files is located in the root of the ESPEI directory as `input-schema.yaml` and is validated in the `espei_script.py` module by the Cerberus package.

14.1.2 Module Hierarchy

- `espei_script.py` is the main entry point for the YAML input API.
- `optimzers` is a package that defines an `OptimizerBase` class for writing optimizers. `EmceeOptimzer` and `ScipyOptimizer` subclasses this.

- `error_functions` is a package with modules for each type of likelihood function.
- `paramselect.py` is where parameter generation happens.
- `mcmc.py` creates the likelihood function and runs MCMC. Deprecated. In the future, users should use `EmceeOptimizer`.
- `parameter_selection` is a package with core pieces of parameter selection.
- `utils.py` are utilities with reuse potential across several parts of ESPEI.
- `plot.py` holds plotting functions.
- `datasets.py` manages validating and loading datasets into a TinyDB in memory database.
- `core_utils.py` are legacy utility functions that should be refactored out to be closer to individual modules and packages where they are used.

14.1.3 Parameter selection

Parameter selection goes through the `generate_parameters` function in the `espei.paramselect` module. The goal of parameter selection is go through each phase (one at a time) and fit a CALPHAD model to the data.

For each phase, the endmembers are fit first, followed by binary and ternary interactions. For each individual endmember or interaction to fit, a series of candidate models are generated that have increasing complexity in both temperature and interaction order (an L0 excess parameter, L0 and L1, ...).

Each model is then fit by `espei.parameter_selection.selection.fit_model`, which currently uses a simple pseudo-inverse linear model from scikit-learn. Then the tradeoff between the goodness of fit and the model complexity is scored by the AICc in `espei.parameter_selection.selection.score_model`. The optimal scoring model is accepted as the model with the fit model parameters set as degrees of freedom for the MCMC step.

The main principle is that ESPEI transforms the data and candidate models to vectors and matricies that fit a typical machine learning type problem of $Ax = b$. Extending ESPEI to use different or custom models in the current scheme basically comes down to formulating candidate models in terms of this type of problem. The main ways to improve on the fitting or scoring methods used in parameter selection is to override the fit and score functions.

Currently the capabilities for providing custom models or contributions (e.g. magnetic data) in the form of generic pycalphad Model objects are limited. This is also true for custom types of data that one would use in fitting a custom model.

14.1.4 MCMC optimization and uncertainty quantification

Most of the Markov chain Monte Carlo optimization and uncertainty quantification happen in the `espei.optimizers.opt_mcmc.py` module through the `EmceeOptimizer` class.

`EmceeOptimizer` is a subclass of `OptimizerBase`, which defines an interface for performing opitmizations of parameters. It defines several methods:

`fit` takes a list of symbol names and datasets to fit to. It calls an `_fit` method that returns an `OptNode` representing the parameters that result from the fit to the datasets. `fit` evaluates the parameters by calling the objective function on some parameters (an array of values) and a context in the `predict` method, which is overridden by `OptimizerBase` subclasses. There is also an interface for storing a history of successive fits to different parameter sets, using the `commit` method, which will store the history of the calls to `fit` in a graph of fitting steps. The idea is that users can generate a graph of fitting results and go back to specific points on the graph and test fitting different sets of parameters or different datasets, creating a unique history of committed parameter sets and optimization paths, similar to a history in version control software like git.

The main reason ESPEI's parameter selection and MCMC routines are split up is that custom Models or existing TDB files can be provided and fit. In other words, if you are using a model that doesn't need parameter selection or is for a property that is not Gibbs energy, MCMC can fit it with uncertainty quantification.

The general process is

1. Take a database with degrees of freedom as database symbols named `VV####`, where `####` is a number, e.g. `0001`. The symbols correspond to `FUNCTION` in the TDB files.
2. Initialize those degrees of freedom to a starting distribution for ensemble MCMC. The starting distribution is controlled by the `EmceeOptimizer.initialize_new_chains` function, which currently supports initializing the parameters to a Gaussian ball.
3. Use the emcee package to run ensemble MCMC

ESPEI's MCMC is quite flexible for customization. To fit a custom model, it just needs to be read by pycalphad and have correctly named degrees of freedom (`VV####`).

To fit an existing or custom model to new types of data, just write a function that takes in datasets and the parameters that are required to calculate the values (e.g. pycalphad Database, components, phases, ...) and returns the error. Then override the `EmceeOptimizer.predict` function to include your custom error contribution. There are examples of these functions `espei.error_functions` that ESPEI uses by default.

Modifications to how parameters are initialized can be made by subclassing `EmceeOptimizer.initialize_new_chains`. Many other modifications can be made by subclassing `EmceeOptimizer`.

Part V

Appendix

CHAPTER 15

Appendices

15.1 Getting Help

For help on installing and using ESPEI, please join the PhasesResearchLab/ESPEI Gitter room.

Bugs and software issues should be reported on GitHub.

15.2 License

ESPEI is MIT licensed.

```
The MIT License (MIT)

Copyright (c) 2015-2018 Richard Otis
Copyright (c) 2017-2018 Brandon Bocklund
Copyright (c) 2018-2019 Materials Genome Foundation

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

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

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

(continues on next page)

(continued from previous page)

```
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

15.3 Citing ESPEI

If you use ESPEI for work presented in a publication, we ask that you cite the following publication:

B. Bocklund, R. Otis, A. Egorov, A. Obaied, I. Roslyakova, Z.-K. Liu, ESPEI for efficient thermodynamic database development, modification, and uncertainty quantification: application to Cu–Mg, MRS Commun. (2019) 1–10. doi:10.1557/mrc.2019.59.

```
@article{Bocklund2019ESPEI,
        archivePrefix = {arXiv},
        arxivId = {1902.01269},
        author = {Bocklund, Brandon and Otis, Richard and Egorov, Aleksei and Obaied,
↪ Abdulmonem and Roslyakova, Irina and Liu, Zi-Kui},
        doi = {10.1557/mrc.2019.59},
        eprint = {1902.01269},
        issn = {2159-6859},
        journal = {MRS Communications},
        month = {jun},
        pages = {1--10},
        title = {{ESPEI for efficient thermodynamic database development,␣
↪modification, and uncertainty quantification: application to Cu-Mg}},
        year = {2019}
}
```

Python Module Index

e

Index

F

G

I

L

M

O

P

R

S

T

U

V