ESPEI Documentation

Release 0.1.3

Brandon Bocklund

Feb 05, 2018

Contents

ESPEI, or Extensible Self-optimizing Phase Equilibria Infrastructure, is a tool for automated thermodynamic database development within the CALPHAD method.

The ESPEI package is based on a fork of pycalphad-fitting and uses pycalphad for calculating Gibbs free energies of thermodynamic models. 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. Single-phase and multi-phase fitting methods are described in Chapter 3 of Richard Otis's thesis.

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).

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).

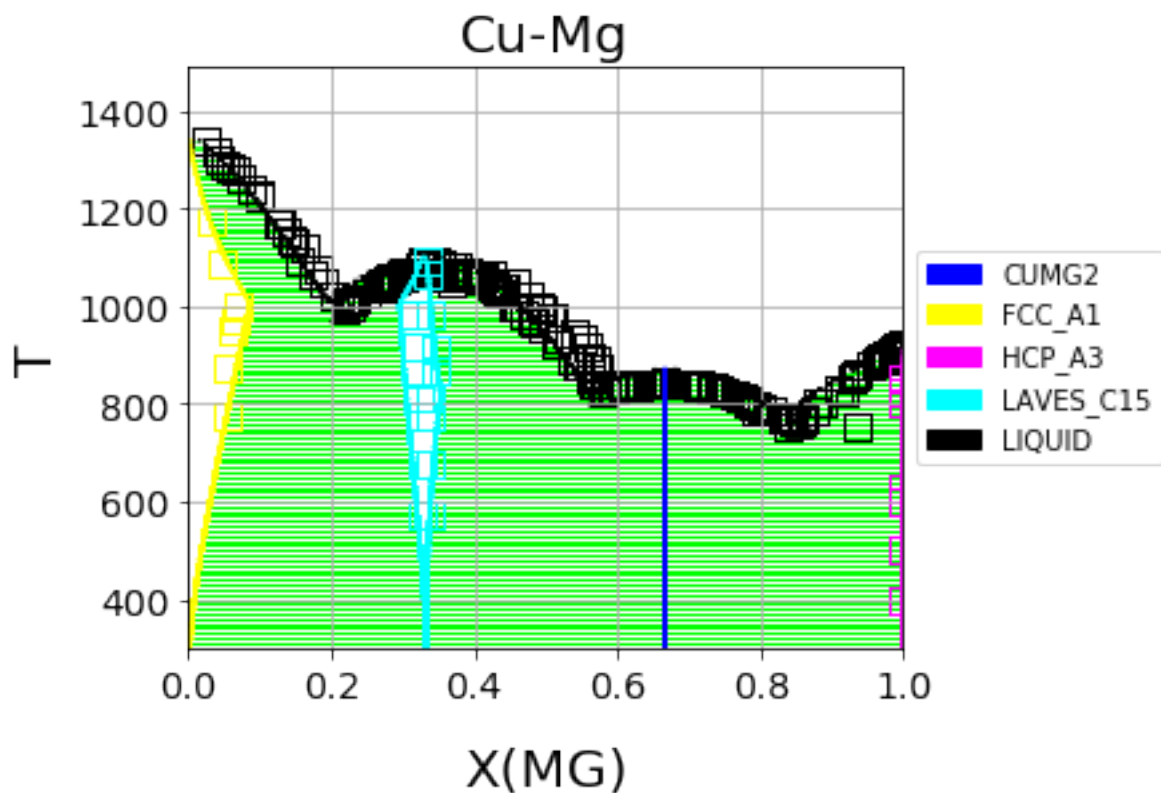Fig. 1: Cu-Mg phase diagram from a database created with and optimized by ESPEI. See the *Cu-Mg Example*.

Installation

1.1 Anaconda (recommended)

ESPEI does not require any special compiler, but several dependencies do. Therefore it is suggested to install ESPEI from conda-forge.

```
conda install -c pycalphad -c msys2 -c conda-forge --yes espei
```

1.2 PyPI

Before you install ESPEI via PyPI, be aware that pycalphad and emcee must be compiled and pycalphad requires an external dependency of Ipopt.

```
pip install espei
```

1.3 Development versions

You may install ESPEI however you like, but here we suggest using Anaconda to download all of the required dependencies. This method installs ESPEI with Anaconda, removes specifically the ESPEI package, and replaces it with the package from GitHub.

```
git clone https://github.com/phasesresearchlab/espei.git
cd espei
conda install espei
conda remove --force espei
pip install -e .
```

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

Usage

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 *Gathering input data* 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 chain (defaults to `chain.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:
  mcmc_steps: 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 chain file called `my-previous-chain.npy`, then you can resume the calculation with the following input file

```
system:
  phase_models: my-phases.json
  datasets: my-input-data
mcmc:
  mcmc_steps: 1000
  input_db: my-tdb.tdb
  restart_chain: my-previous-chain.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:
  mcmc_steps: 1000
```

2.5 Input Customization

ESPEI lets you control many aspects of your calculations with the input files shown above. See *Writing ESPEI input* 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 `chain.npy` and `lnprob.npy` for the MCMC chain at the end of your run and according to the save interval (defaults to every 20 iterations). 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. To use ESPEI with MPI, you simply call ESPEI in the same way as above with *mpirun* or whichever MPI software you use. You also must indicate to ESPEI that it should create an MPI scheduler by setting the input option `scheduler: MPIPool` in the `mcmc` heading. Be aware that `mpi4py` must be compiled with an MPI-enabled compiler, see the mpi4py installation instructions.

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.

Module Hierarchy

- `fit.py` is the main entry point
- `paramselect.py` is where all of the fitting happens. This is the core.
- `core_utils.py` contains specialized utilities for ESPEI.
- `utils.py` are utilities with reuse potential outside of ESPEI.
- `plot.py` holds plotting functions

License

ESPEI is MIT licensed. See LICENSE.

5.1 What's New

5.1.1 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

5.1.2 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)

5.1.3 0.3.1.post2 (2017-10-31)

- Propogate the new entry point to setup.py

5.1.4 0.3.1.post1 (2017-10-31)

- Fix for module name/function conflict in entry point

5.1.5 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

5.1.6 0.3.post2 (2017-09-20)

- Add input-schema.yaml file to installer

5.1.7 0.3.post1 (2017-09-20)

- Add LICENSE to manifest

5.1.8 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

5.1.9 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.

5.1.10 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 explictly 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 compatibiltiy
    - unicode datasets
    - handling of singular matrix errors from pycalphad's `equilibrium`
    - reporting of failed conditions

5.1.11 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

5.1.12 0.1.4 (2017-07-24)

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

5.1.13 0.1.3 (2017-06-23)

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

5.1.14 0.1.2 (2017-06-23)

- Fix dask incompatibilty due to new API usage

5.1.15 0.1.1 (2017-06-23)

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

5.1.16 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

5.2 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], Zou 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.

---

[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

5.2.1 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 *Gathering input data* 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.

5.2.2 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

```json
{
  "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"]]
        }
    }
}
```

5.2.3 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 *Gathering input data*, 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.

5.2.4 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.

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

```
# 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)
```

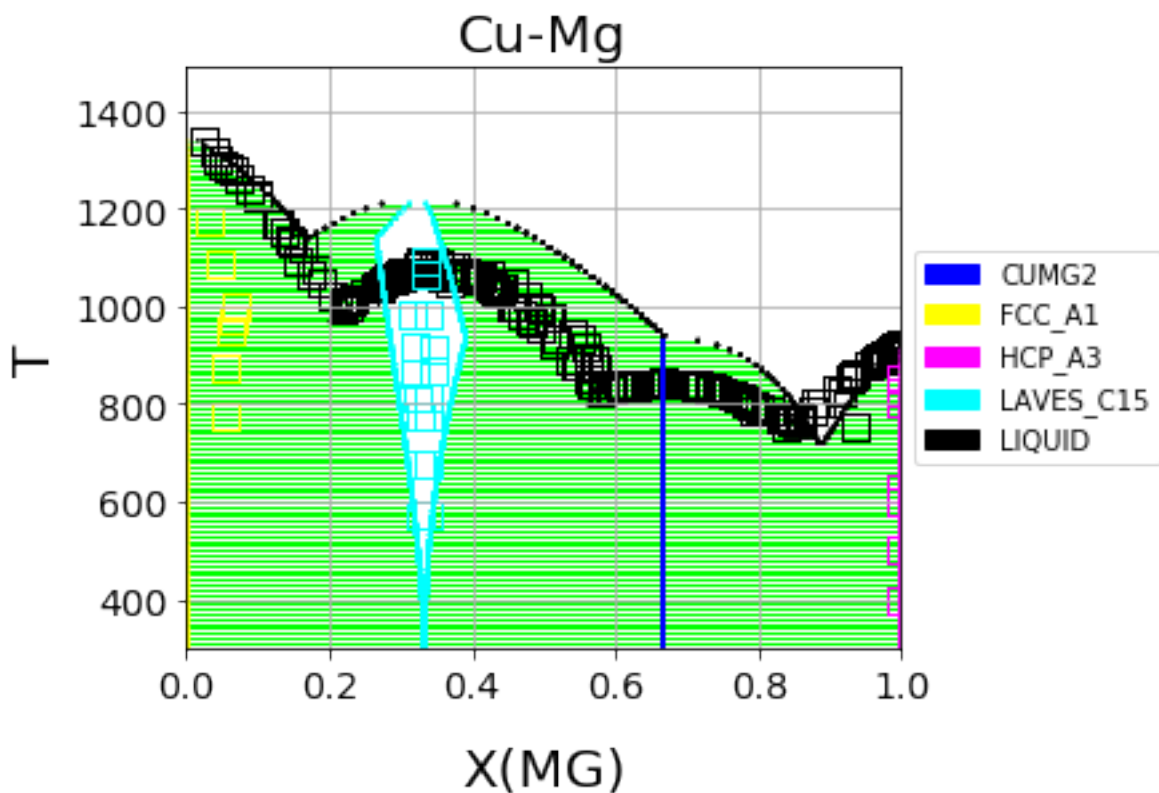Which should result in the following figure

Fig. 5.1: 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.

5.2.5 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. For more details on how MCMC is used in ESPEI, see the *Theory* page.

For the Cu-Mg system, it has been experimentally reported that Cu has no solubility in Mg, so there are few measurements of solubility reported in the literature. In order to properly reproduce this and prevent other parameters to be optimized in a way that introduces solubility in the hcp phase, we have added the phase equilibria for the hcp phase (pink squares) to have 100% Mg (0% Cu). These points effectively anchor the hcp phase to have no solubility. Because thermodynamic databases are typically developed for pragmatic reasons, it is sensible to use these estimates, even for more complicated equilibria that there is no data available for. ESPEI allows thermodynamic database 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.

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:
  mcmc_steps: 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 steps will never be accepted (they return a log-probability of $-\infty$).

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

```python
# 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

# 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']
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)
```
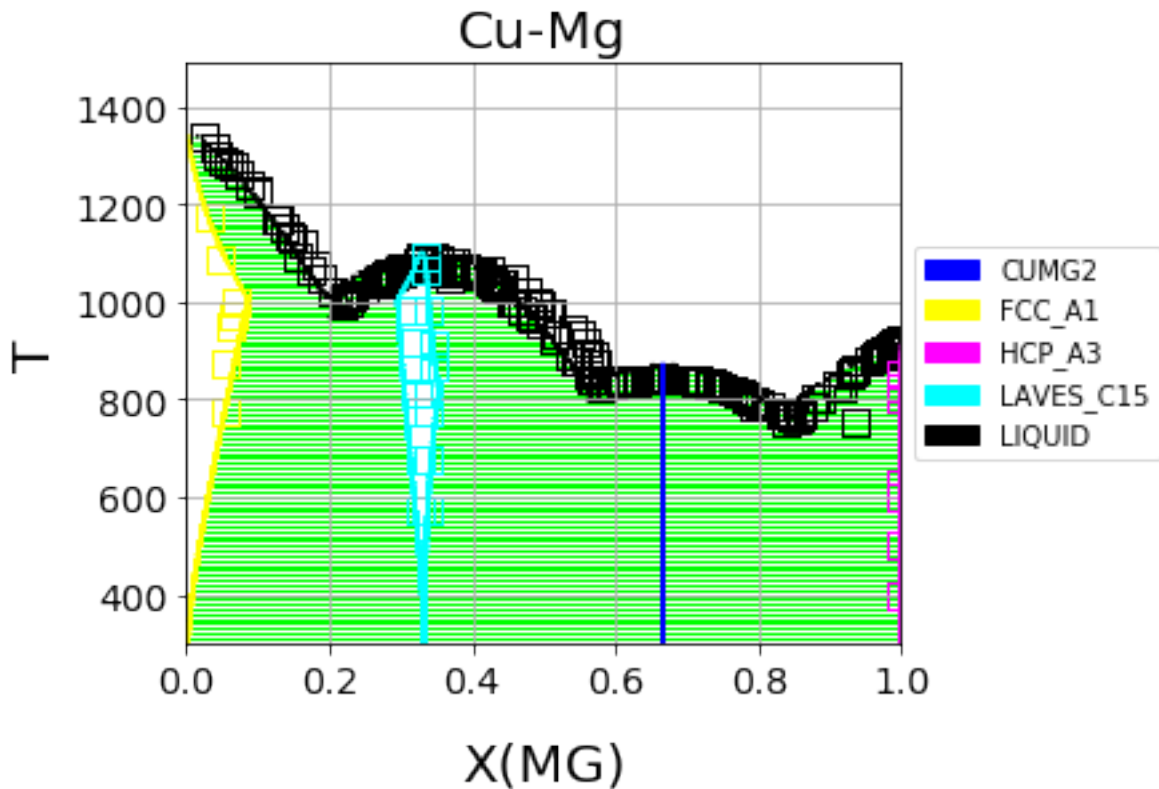

Fig. 5.2: Optimized Cu-Mg phase diagram from the multi-phase fitting in ESPEI

5.2.6 References

5.2.7 Acknowledgements

Credit for initially preparing the datasets goes to Aleksei Egorov.

5.3 Gathering input data

5.3.1 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`.

5.3.2 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"]]
        }
    }
}
```

5.3.3 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. 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. The formatting of this can be tricky, and it is suggested to use a NumPy array and reshape or add axes using `np.newaxis` indexing.

```
{
  "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)"
  },
```

```
    "conditions": {
            "P": 101325,
            "T": [  0,  10,  20,  30,  40,  50,  60,  70,  80,  90, 100, 110]
    },
    "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"],
                                      ["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]]]
}
```

5.3.4 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"
}
```

5.4 Writing ESPEI input

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

output:
  verbosity
  output_db
  tracefile
  probfile

generate_parameters:
  excess_model
  ref_state

mcmc:
  mcmc_steps
  mcmc_save_interval
  cores
  scheduler
  input_db
  restart_chain
  chains_per_parameter
  chain_std_deviation
  deterministic
```

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

5.4.1 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 *Gathering input data*.

5.4.2 output

verbosity

> **type** int
>
> **default** 0

Controls the logging level.

Value	Log Level
0	Warning
1	Info
2	Debug

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** chain.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 steps, 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.

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 steps, 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.

5.4.3 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.

5.4.4 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').

mcmc_steps

> **type** int
>
> **default** required

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

mcmc_save_interval

> **type** int
>
> **default** 20

Controls the interval for saving the MCMC chain and probability files.

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** emcee
>
> **options** emcee | MPIPool | dask | None

Which scheduler to use for parallelization. You can choose from either *dask*, *emcee*, *MPIPool* or *None*.

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

Choosing MPIPool will allow you to set the number of cores directly using MPI.

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

It is recommended to use MPIPool if you will be running jobs on supercomputing clusters.

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_chain

> **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_chain`) 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 steps. If these are both restarted after 100 steps for another 50 steps, then the final chain after 150 steps 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 steps while Run 2 proceeds 100 steps, restarting Run 1 for 100 steps and Run 2 for 50 steps (so they are both at 150 total steps) will **NOT** give the same result.

5.5 Theory

ESPEI has two types of fitting – parameter generation and MCMC optimization. The parameter generation step uses experimental and DFT data of the Gibbs free energy derivatives (C_P, H, S) for each phase and for the mixing energies within sublattices for each phase to generate and fit parameters of given CALPHAD models. The MCMC optimization step uses a Bayesian optimization procedure to fit parameters in a Database to experimental phase equilibria.

5.5.1 Parameter generation

A simple model with few parameters is better than a complex model that describes the same data marginally better. Parameter generation in ESPEI aims to achieve a balance of a simple parameterization and goodness of fit in the Redlich-Kister polynomial used in CALPHAD assessments. To achieve this, parameters are selected using the Akaike information criterion (AIC) to choose an optimal set of parameters from canditate parameterizations.

The general Redlich Kister polynomial has the form $G = a + bT + cT\ln T + \sum_n d_n T^n$. Different parameterizations, e.g. only considering a, considering a and b, a and c, etc. are fit to all of the input formation or mixing data (depending on the parameter being selected) by a least squares pseudo-inverse optimization.

Each parameterization is compared in the AIC and the most suitable optimization balances the goodness of fit and the number of parameters. The key aspect of this is that ESPEI will avoid overfitting your data and will not add parameters you do not have data for.

This is important for phases that would have a temperature dependent contribution to the Gibbs energy, but the input data only gives 0K formation energies. ESPEI cannot add temperature dependence to the parameterized model. Because of this, an abundance of single-phase data is critical to provide enough degrees of freedom in later optimization.

5.5.2 MCMC optimization

Details of Markov Chain Monte Carlo as an algorithm are better covered elsewhere. A good example is MacKay's (free) book: Information Theory, Inference, and Learning Algorithms.

Using MCMC for optimizing CALPHAD models might appear to have several drawbacks. The parameters in the models are correlated and due to the nature of single phase first-principles data the shape and size of the posterior distribution for each parameter is not known before fitting. Traditional Metropolis-Hastings MCMC algorithms require the a prior to be defined for each parameter, which is a challenge for parameters in CALPHAD models which vary over more than 6 orders of magnitude.

ESPEI solves these potential problems by using an Ensemble sampler, as introduced by Goodman and Weare[1], rather than the Metropolis-Hastings algorithm. Ensemble samplers have the property of affine invariance, which uses multiple ($\geq 2N$ for N parameters) parallel chains to scale new proposal parameters by linear transforms. These chains, together an ensemble, define a proposal distribution to sample parameters from that is scaled to the magnitude and sensitivity of each parameter. Thus, Ensemble samplers directly address the challenges we expect to encounter with traditional MCMC.

ESPEI uses an Ensemble sampler algorithm by using the emcee package that implements parallelizable ensemble samplers. To use emcee, ESPEI defines the initial ensemble of chains and a function that returns the error as a log-probability. ESPEI defines the error as the mean square error between experimental phase equilibria and the equilibria calculated by the CALPHAD database.

Here, again, it is critical to point out the importance of abundant phase equilibria data. Traditional CALPHAD modeling has involved the modeler participating in tight feedback loops between updates to parameters and the resulting phase diagram. ESPEI departs from this by optimizing just a single scalar error function based on phase equilibria. The implication of this is that if there are phase equilibria that are observed to exist, but they are not in the datasets

---

[1] Goodman, J. & Weare, J. Ensemble samplers with affine invariance. Commun. Appl. Math. Comput. Sci. 5, 65–80 (2010). doi:10.2140/camcos.2010.5.65.

that are considered by ESPEI, those equilibria cannot be optimized against and may deviate from 'known' equilibria. A possible approach to address this in ESPEI is to estimate the points for the equilibria.

5.5.3 References

5.6 API Documentation

5.6.1 espei package

Subpackages

espei.tests package

Submodules

espei.tests.fixtures module

Fixtures for use in tests

espei.tests.test_core_utils module

espei.tests.test_core_utils.**test_get_data_for_a_minimal_example**()
  Given a dataset and the congfiguration pertaining to that dataset, we should find the values.

espei.tests.test_datasets module

espei.tests.test_datasets.**test_check_datasets_raises_on_misaligned_data**()
  Passed datasets that have misaligned data and conditions should raise DatasetError.

espei.tests.test_datasets.**test_check_datasets_raises_with_incorrect_components**()
  Passed datasets that have incorrect components entered vs. used should raise.

espei.tests.test_datasets.**test_check_datasets_raises_with_incorrect_zpf_phases**()
  Passed datasets that have incorrect phases entered than used should raise.

espei.tests.test_datasets.**test_check_datasets_raises_with_malformed_sublattice_configuratic**
  Passed datasets that have malformed ZPF values should raise.

espei.tests.test_datasets.**test_check_datasets_raises_with_malformed_zpf**()
  Passed datasets that have malformed ZPF values should raise.

espei.tests.test_datasets.**test_check_datasets_run_on_good_data**()
  Passed valid datasets that should raise DatasetError.

espei.tests.test_mcmc module

espei.tests.test_mcmc.**test_lnprob_calculates_probability_for_success**(*datasets_db*)
  lnprob() successfully calculates the probability for equilibrium

espei.tests.test_mcmc.**test_parameter_initialization**()
  Determinisitically generated parameters should match.

---

espei.tests.test_parameter_generation module

The test_parameter_generation module tests that parameter selection is correct

espei.tests.test_parameter_generation.**test_mixing_energies_are_fit**(*datasets_db*)
> Tests that given mixing energy data, the excess parameter is fit.

espei.tests.test_parameter_generation.**test_sgte_reference_state_naming_is_correct_for_chara**
> Elements with single character names should get the correct GHSER reference state name (V => GHSERVV)

espei.tests.test_schema module

Tests for input file validation

espei.tests.test_schema.**test_SR2016_refdata**()

espei.tests.test_schema.**test_chains_per_parameter_read_correctly**()
> The chains per parameter option should take effect when passed.

espei.tests.test_schema.**test_correct_defaults_are_applied_from_minimal_specification**()
> A minimal run should generate several default settings for i/o and optional settings.

espei.tests.test_schema.**test_input_yaml_invalid_for_mcmc_when_input_is_overspecified**()
> An MCMC run must get input from only generate_parameters or an input tdb (w/ or w/o a restart).

espei.tests.test_schema.**test_input_yaml_invalid_for_mcmc_when_input_not_defined**()
> An MCMC run must get input from generate_parameters, an input tdb, or a restart and input tdb.

espei.tests.test_schema.**test_input_yaml_valid_for_full_run**()
> A minimal full run input file should validate

espei.tests.test_schema.**test_input_yaml_valid_for_generate_parameters_only**()
> A minimal generate parameters only input file should validate

espei.tests.test_schema.**test_input_yaml_valid_for_mcmc_from_restart**()
> A minimal mcmc run from a restart should validate

espei.tests.test_schema.**test_input_yaml_valid_for_mcmc_from_tdb**()
> A minimal mcmc run from tdb input file should validate

espei.tests.test_utils module

Test espei.utils classes and functions.

espei.tests.test_utils.**test_adding_bibtex_entries_to_bibliography_db**(*datasets_db*)
> Adding a BibTeX entries to a database works and the database can be searched.

espei.tests.test_utils.**test_bib_marker_map**()
> bib_marker_map should return a proper dict

espei.tests.test_utils.**test_flexible_open_string_file_like**(*tmp_file*)
> File-like objects support read methods should have their content returned by flexible_open_string.

espei.tests.test_utils.**test_flexible_open_string_path_like**(*tmp_file*)
> Path-like strings should be opened, read and returned

espei.tests.test_utils.**test_flexible_open_string_raw_string**()
> Raw multiline strings should be directly returned by flexible_open_string.

`espei.tests.test_utils.`**`test_immediate_client_returns_map_results_directly`**`()`
> Calls ImmediateClient.map should return the results, instead of Futures.

`espei.tests.test_utils.`**`test_pickelable_tinydb_can_be_pickled_and_unpickled`**`()`
> PickleableTinyDB should be able to be pickled and unpickled.

Module contents

Submodules

espei.core_utils module

Module for handling data

`espei.core_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.core_utils.`**`canonical_sort_key`**`(`*x*`)`
> Wrap strings in tuples so they'll sort.
>
> > **Parameters** **x** (`[str]`) – list of strings
> >
> > **Returns** tuple of strings that can be sorted
> >
> > **Return type** (str)

`espei.core_utils.`**`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.core_utils.`**`endmembers_from_interaction`**`(`*configuration*`)`

`espei.core_utils.`**`get_data`**`(`*comps*, *phase_name*, *configuration*, *symmetry*, *datasets*, *prop*`)`

`espei.core_utils.`**`get_samples`**`(`*desired_data*`)`

`espei.core_utils.`**`list_to_tuple`**`(`*x*`)`

`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

> Depdendent 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.`**`symmetry_filter`**(*x*, *config*, *symmetry*)

> Return True if the candidate sublattice configuration has any symmetry which matches the phase model symmetry.

> > **Parameters**

- **x** (*the candidate dataset 'solver' dict. Must contain the* *"sublattice_configurations" key*) –

- **config** (*the configuratino of interest: e.g. ['AL', ['AL',* *'NI'], 'VA']*) –

- **symmetry** (*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.**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

> Planned validation checks: * all required keys are present * individual shapes of keys, such as ZPF, sublattice configs and site ratios

> 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.**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*)

> 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_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.`**`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

Module for running MCMC in ESPEI

`espei.mcmc.`**`estimate_hyperplane`**(*dbf*, *comps*, *phases*, *current_statevars*, *comp_dicts*, *phase_models*, *parameters*)

`espei.mcmc.`**`generate_parameter_distribution`**(*parameters*, *num_samples*, *std_deviation*, *deterministic=True*)
> Return an array of num_samples from a Gaussian distribution about each parameter.

>> **Parameters**

>>> • **`parameters`** (*ndarray*) – 1D array of initial parameters that will be the mean of the distribution.

>>> • **`num_samples`** (*int*) – Number of chains to initialize.

>>> • **`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

`espei.mcmc.`**`lnprob`**(*params*, *comps=None*, *dbf=None*, *phases=None*, *datasets=None*, *symbols_to_fit=None*, *phase_models=None*, *scheduler=None*)
> Returns the error from multiphase fitting as a log probability.

`espei.mcmc.`**`mcmc_fit`**(*dbf*, *datasets*, *mcmc_steps=1000*, *save_interval=100*, *chains_per_parameter=2*, *chain_std_deviation=0.1*, *scheduler=None*, *tracefile=None*, *probfile=None*, *restart_chain=None*, *deterministic=True*)
> Run Markov Chain Monte Carlo on the Database given datasets

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 to fit

- **mcmc_steps** (`int`) – Number of chain steps to calculate in MCMC. Note the flattened chain will have (mcmc_steps*DOF) values. Default is 1000 steps.

- **save_interval** (`int`) – interval of steps to save the chain to 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 flattened chain with NumPy.save. Array has shape (nwalkers, iterations, nparams)

- **probfile** (`str`) – filename to store the flattened ln probability with NumPy.save

- **restart_chain** (`np.ndarray`) – ndarray of the previous chain. Should have shape (nwalkers, iterations, nparams)

- **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_chain) will produce exactly the same results.

Returns

- **dbf** (*Database*) – Resulting pycalphad database of optimized parameters

- **sampler** (*EnsembleSampler, ndarray*)) – emcee sampler for further data wrangling

espei.mcmc.**multi_phase_fit**(*dbf*, *comps*, *phases*, *datasets*, *phase_models*, *parameters=None*, *scheduler=None*)

espei.mcmc.**tieline_error**(*dbf*, *comps*, *current_phase*, *cond_dict*, *region_chemical_potentials*, *phase_flag*, *phase_models*, *parameters*, *debug_mode=False*)

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 AIC 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 AIC score.

4. G (full MCMC) - all parameters selected at least once by above procedure

MCMC uses an EnsembleSampler based on Goodman and Weare, Ensemble Samplers with Affine Invariance. Commun. Appl. Math. Comput. Sci. 5, 65-80 (2010).

`espei.paramselect.`**`fit_formation_energy`**(*dbf*, *comps*, *phase_name*, *configuration*, *symmetry*, *datasets*, *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.

> > > - **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.`**`generate_parameters`**(*phase_models*, *datasets*, *ref_state*, *excess_model*)

> 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'

> > **Returns**

> > **Return type** Database

`espei.paramselect.`**`phase_fit`**(*dbf*, *phase_name*, *symmetry*, *subl_model*, *site_ratios*, *datasets*, *refdata*, *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

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

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

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

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) # plot the mixing interaction in the first sublattice >>> plot_parameters(dbf, ['CU', 'MG'], 'LAVES_C15', (('CU', 'MG'), 'MG'), symmetry=None, datasets=datasets)

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.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

```
>>> bib_marker_map(['otis2016', 'bocklund2018'])
{
'bocklund2018': {
            'formatted': 'bocklund2018',
            'markers': {'fillstyle': 'full', 'marker': 'o'}
        },
'otis2016': {
            'formatted': 'otis2016',
```

```
                'markers': {'fillstyle': 'full', 'marker': 'v'}
        }
}
```

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**

> > **Return type** list

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

Module contents

5.7 Contributing to ESPEI

5.7.1 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
```

5.7.2 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

5.7.3 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.

5.7.4 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 steps.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

e

Index

S

T