ESPEI Documentation

Release 0.1.3

Brandon Bocklund

Aug 02, 2017

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).

CHAPTER 1

Installation

Creating a virual environment is highly recommended. You can install ESPEI from PyPI

```
pip install espei
```

or install in develop mode from source

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

CHAPTER 2

Usage

Run `espei -h` to see the options in the command utility.

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 fit settings 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 to fit. 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 *Writing input files* 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`) and an array of the steps in the MCMC chain (defaults to `chain.txt`).

Full run

A minimal run of ESPEI with single phase fitting and MCMC fitting would involve setting these two files

```
espei --datasets=my-dataset-folder --fit-settings=my-input.json
```

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. To do this, simply pass the `--no-mcmc` flag to ESPEI

```
espei --no-mcmc --datasets=my-dataset-folder --fit-settings=my-input.json
```

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 with

```
espei --datasets=my-dataset-folder --fit-settings=my-input.json --input-tdb=my-
↪starting-database.tdb
```

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

Customization

In all cases, ESPEI lets you control certain aspects of your calculations from the command line. Some useful options are

- `verbose` (or `-v`) controls the logging level. Default is Warning. Using verbose once gives more detail (Info) and twice even more (Debug)
- `tracefile` lets you set the output trace of the chain to any name you want. The default is `chain.txt`.
- `output-tdb` sets the name of the TDB output at the end of the run. Default is `out.tdb`.
- `input-tdb` is for setting input TDBs. This will skip single phase fitting and fit all parameters defined as FUNCTIONs with names starting with `VV`.
- `no-mcmc` will do single-phase fitting only. Default is to perform MCMC fitting.
- `mcmc-steps` sets the number of MCMC steps. The default is 1000.
- `save-interval` controls the interval for saving the MCMC chain. The default is 100 steps.

Run `espei -h` to see all of the configurable options.

FAQ

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.

To find the offending files, you can rename the datasets to anything not ending in `.json`, such as `my_datasets.json.disabled`. The renamed files will be ignored and it allows you to track down any problematic files.

CHAPTER 3

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

CHAPTER 4

License

ESPEI is MIT licensed. See LICENSE.

Writing input files

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`.

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"]]
      }
    }
}
```

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]]]
}
```

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"
}
```

What's New

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

0.1.4 (2017-07-24)

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

0.1.3 (2017-06-23)

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

0.1.2 (2017-06-23)

- Fix dask incompatibilty due to new API usage

0.1.1 (2017-06-23)

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

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

API Documentation

espei package

Subpackages

espei.tests package

Submodules

espei.tests.test_datasets module

espei.tests.test_utils module

Test espei.utils classes and functions.

`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**()

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.**symmetry_filter**(*x*, *config*, *symmetry*)

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`** –
- **`--------`** –
- **`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.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.`**`estimate_hyperplane`**(*dbf*, *comps*, *phases*, *current_statevars*, *comp_dicts*, *phase_models*, *parameters*)

`espei.paramselect.`**`fit`**(*input_fname*, *datasets*, *resume=None*, *scheduler=None*, *run_mcmc=True*, *tracefile=None*, *mcmc_steps=1000*, *save_interval=100*)

Fit thermodynamic and phase equilibria data to a model.

Parameters

- **`input_fname`** (*`str`*) – name of the input file containing the sublattice models.
- **`datasets`** (*`PickleableTinyDB`*) – database of single- and multi-phase to fit.
- **`resume`** (*`Database`*) – pycalphad Database of a file to start from. Using this parameter causes single phase fitting to be skipped (multi-phase only).
- **`scheduler`** (*`callable`*) – Scheduler to use with emcee. Must implement a map method.
- **`run_mcmc`** (*`bool`*) – Controls if MCMC should be run. Default is True. Useful for first-principles (single-phase only) runs.
- **`tracefile`** (*`str`*) – filename to store the flattened chain with NumPy.savetxt
- **`mcmc_steps`** (*`int`*) – number of chain steps to calculate in MCMC. Note the flattened chain will have (mcmc_steps*DOF) values. int (Default value = 1000)
- **`save_interval`** (*`int`*) – interval of steps to save the chain to the tracefile.

Returns

- **dbf** (*Database*) – Resulting pycalphad database of optimized parameters
- **sampler** (*EnsembleSampler, ndarray)*) – emcee sampler for further data wrangling
- **parameters_dict** (*dict*) – Optimized parameters

`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.`**`lnprob`**(*params*, *data=None*, *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.paramselect.`**`multi_phase_fit`**(*dbf*, *comps*, *phases*, *datasets*, *phase_models*, *parameters=None*, *scheduler=None*)

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

espei.plot module

Plotting of input data and calculated database quantities

`espei.plot.`**`multi_plot`**(*dbf*, *comps*, *phases*, *datasets*, *ax=None*)

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

espei.run_espei module

Automated fitting script.

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

`espei.run_espei.`**`main`**()

espei.utils module

Utilities for ESPEI

Classes and functions defined here should have some reuse potential.

`espei.utils.`**`sigfigs`**`(`*x*, *n*`)`

Round x to n significant digits

Module contents

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index