ESPEI Documentation

Release 0.1.3

Brandon Bocklund

Oct 31, 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).

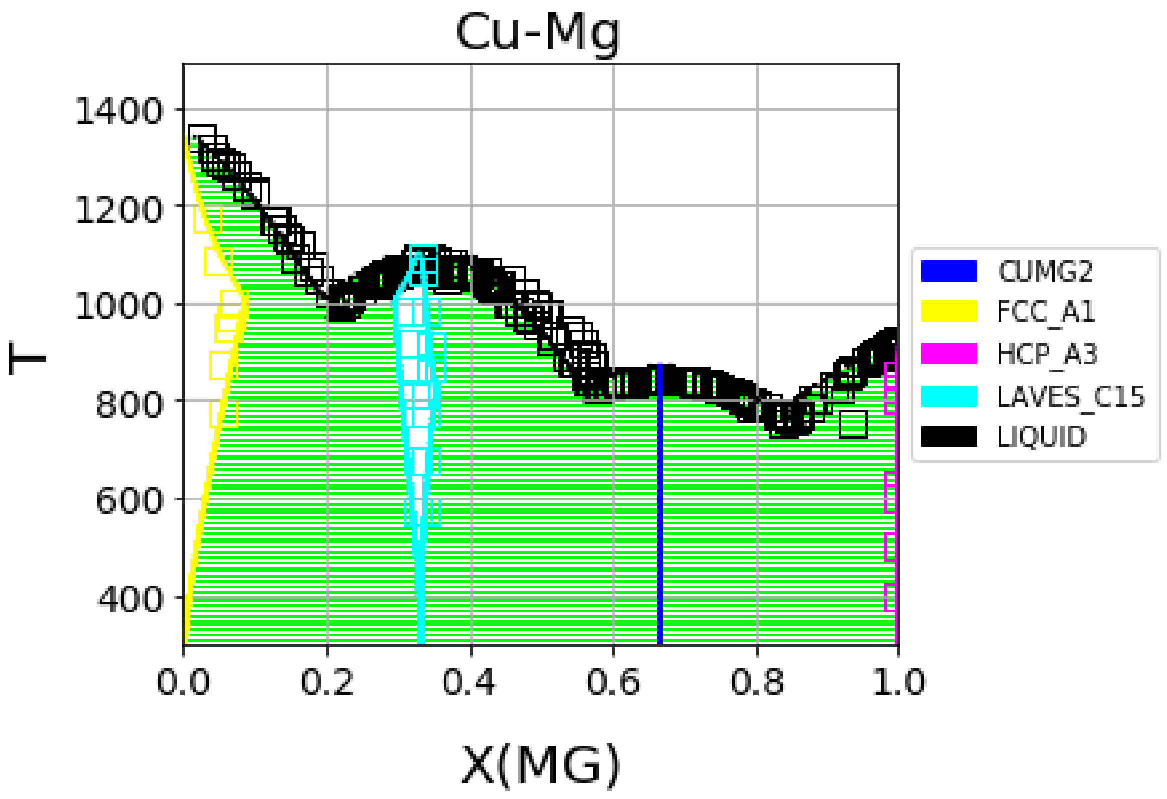

Fig. 1: Cu-Mg phase diagram from a database created with and optimized by ESPEI. See the *Cu-Mg Example*.

CHAPTER 1

Installation

Creating a virual environment is highly recommended. ESPEI does not require any special compiler, but several dependencies do. Therefore it is suggested to install ESPEI from conda-forge

```
conda config --add channels conda-forge
conda create -n my_env espei
```

Alternatively, ESPEI is available 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

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`).

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
```

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`.

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
```

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
```

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.

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.

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

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.

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.

CHAPTER 3

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.

CHAPTER 4

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 5

License

ESPEI is MIT licensed. See LICENSE.

What's New

0.3.1.post2 (2017-10-31)

- Propogate the new entry point to setup.py

0.3.1.post1 (2017-10-31)

- Fix for module name/function conflict in entry point

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

0.3.post2 (2017-09-20)

- Add input-schema.yaml file to installer

0.3.post1 (2017-09-20)

- Add LICENSE to manifest

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

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.

0.2 (2017-08-15)

- New `multiplot` interface for convienent 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

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

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

[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

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.

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.

Phases and CALPHAD models

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

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

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

```
{
  "components": ["CU", "MG", "VA"],
  "refdata": "SGTE91",
  "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"]]
         },
```

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

```
            "LAVES_C15": {
               "sublattice_site_ratios": [2, 1],
               "sublattice_model": [["CU", "MG"], ["CU", "MG"]]
            }
      }
}
```

Of note is that we will be using the reference state defined by SGTE91. This reference state is implemented in `pycalphad.refdata`. Other reference states can be supported in principle, but must be implemented in `pycalphad.refdata`.

ESPEI

ESPEI has two steps: single-phase fitting and multi-phase fitting. The single-phase fitting step uses experimental and DFT data for derivatives of the Gibbs free energy (C_P, H, S) for a phase and for the mixing energies within sublattices for each phase to select and fit parameters. The multi-phase fitting uses MCMC to optimize the parameters selected and calculated from the single-phase fitting. These steps can be performed together, or separately. For clarity, we will preform these steps separately. The next two sections are devoted to describing ESPEI's single-phase fitting and MCMC 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.

First-principles phase diagram

In single-phase fittings, parameters are selected using the Akaike information criterion (AIC) to choose an optimal set of parameters from canditate models (here, parameters in Redlich-Kister polynomials) that balance the number of parameters with the goodness of fit. The AIC prevents overfitting or underfitting Gibbs free energy functions with parameterizations with data. 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 in the case of structures that should have some temperature dependent contribution to the Gibbs energy parameterization, but the input data only gives 0K formation energies. In this case, temperature dependence cannot be added and calculated to the thermodynamic model. Thus, an abundance of single-phase data is critical to provide enough degrees of freedom in later optimization.

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

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

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

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

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

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

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

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

# 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

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.

MCMC-based Bayesian optimization

With the data in the CU-MG input data, ESPEI generated 11 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. Details of MCMC are better covered elsewhere, such as MacKay's (free) book: Information Theory, Inference, and Learning Algorithms.

Using MCMC for optimizing CALPHAD models might appear to have several drawbacks. As previously mentioned, the parameters in the models are correlated, but we are also unsure about the shape and size of the posterior distribution

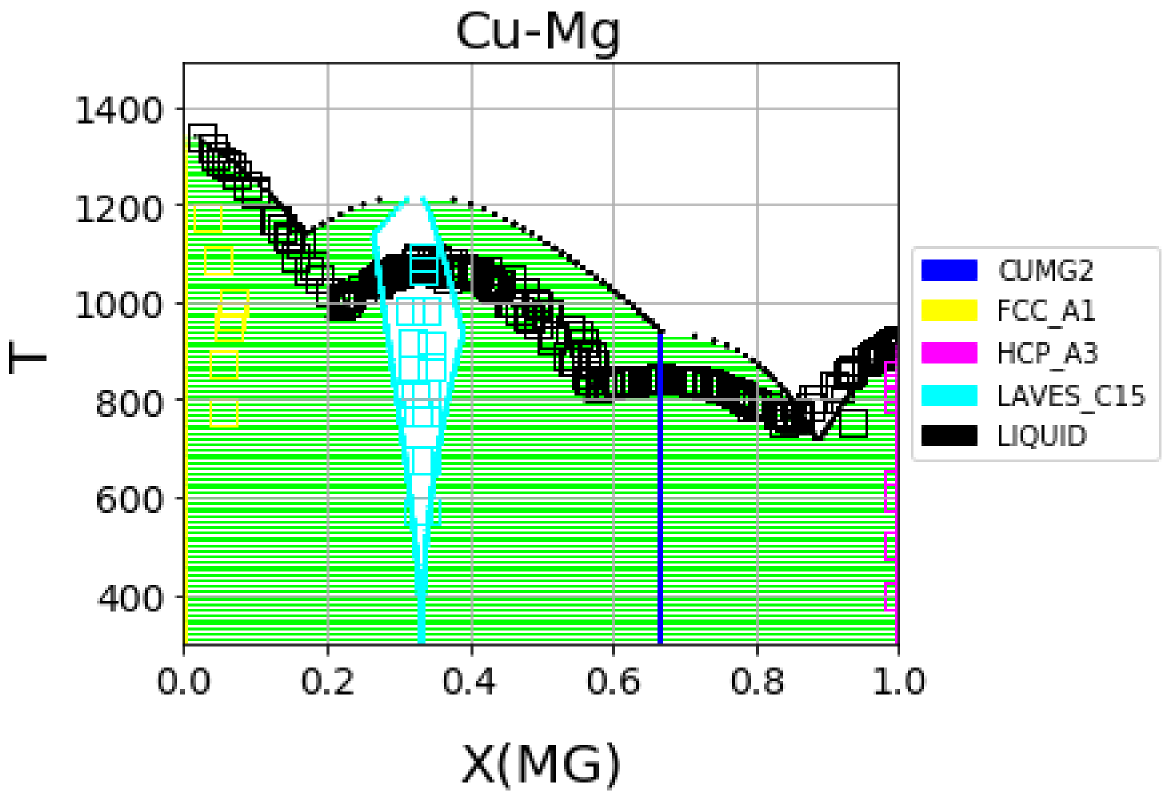

Fig. 5.1: First-principles Cu-Mg phase diagram from the single-phase fitting in ESPEI

for each parameter before fitting. Particularly, traditional Metropolis-Hastings MCMC algorithms require the a prior to be defined for each parameter, which is a problem 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[5], 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 that are considered by ESPEI, those equilibria cannot be optimized against and may deviate from 'known' equilibria.

We address this in ESPEI by estimating points for the equilibria. 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).

[5] 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.

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

```
# 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)
```

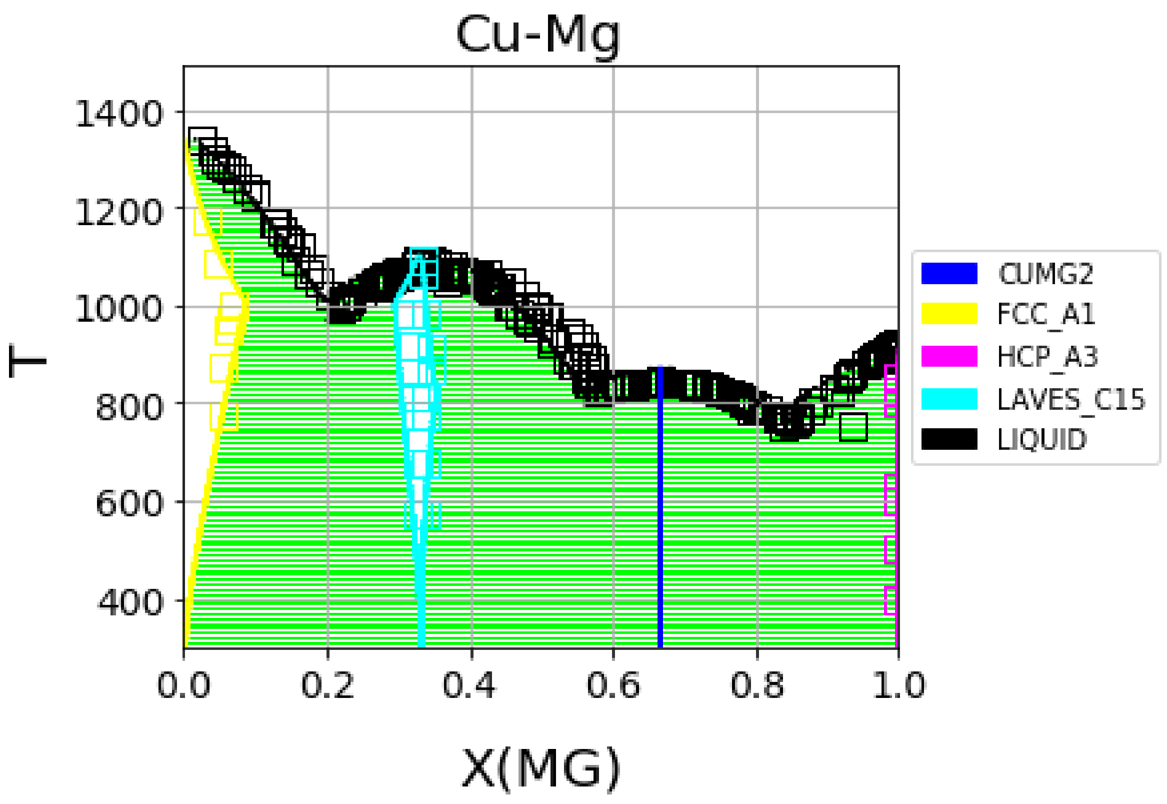

Fig. 5.2: Optimized Cu-Mg phase diagram from the multi-phase fitting in ESPEI

References

Acknowledgements

Credit for initially preparing the datasets goes to Aleksei Egorov.

Gathering input data

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 book-keeping 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"
}
```

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
```

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

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*.

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.

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

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

There are plans to extend to the SGTE Unary 5 and also support custom reference states.

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 dask | emcee | MPIPool

Which scheduler to use for parallelization. You can choose from either *dask*, *emcee*, or *MPIPool*.

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.

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.

API Documentation

espei package

Subpackages

espei.tests package

Submodules

espei.tests.fixtures module

espei.tests.test_core_utils module

espei.tests.test_datasets module

espei.tests.test_parameter_generation module

espei.tests.test_paramselect module

espei.tests.test_schema module

espei.tests.test_utils module

Module contents

Submodules

espei.core_utils module

espei.datasets module

espei.espei_script module

espei.paramselect module

espei.plot module

espei.refdata module

espei.utils module

Module contents

Contributing to ESPEI

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
```

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

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.

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