Journal of
open research software

Otis, R and Liu, Z-K 2017 pycalphad: CALPHAD-based Computational Thermodynamics in Python. *Journal of Open Research Software,* 5: 1, DOI: http://dx.doi.org/10.5334/jors.140

SOFTWARE METAPAPER

# pycalphad: CALPHAD-based Computational Thermodynamics in Python

Richard Otis and Zi-Kui Liu

Department of Materials Science and Engineering, Pennsylvania State University, University Park, PA 16802, US

Corresponding author: Richard Otis (richard.otis@outlook.com)

The pycalphad software package is a free and open-source Python library for designing thermodynamic models, calculating phase diagrams and investigating phase equilibria using the CALPHAD method. It provides routines for reading thermodynamic databases and solving the multi-component, multi-phase Gibbs energy minimization problem. The pycalphad software project advances the state of thermodynamic modeling by providing a flexible yet powerful interface for manipulating CALPHAD data and models. The key feature of the software is that the thermodynamic models of individual phases and their associated databases can be programmatically manipulated and overridden at run-time without modifying any internal solver or calculation code. Because the models are internally decoupled from the equilibrium solver and the models themselves are represented symbolically, pycalphad is an ideal tool for CALPHAD database development and model prototyping.

**Keywords:** CALPHAD; computational thermodynamics; alloys; Python
**Funding Statement:** This work was supported by a NASA Space Technology Research Fellowship under grant number NNX14AL43H.

## (1) Overview

### Introduction

Thermodynamics is the core of every physical description of nature. In recognition of this fact, and coincident with the rise of ubiquitous modern computing, the development of the CALculation of PHAse Diagrams (CALPHAD) method was proposed by Larry Kaufman and Himo Ansara in 1973 to rationalize and systematize alloy chemistry through the use of computer calculations [1]. In the decades since, there has been a tremendous effort by the scientific community to collect data to build thermodynamic descriptions for both metallic and non-metallic systems, descriptions of which have only increased in sophistication and accuracy as our understanding of the underlying physical phenomena have improved.

Better understanding of CALPHAD modeling and equilibrium calculation are necessary for advancing the modeling of phase stability in alloy systems and, for practical materials design problems, we need to have a high-quality software implementation flexible enough to admit these theoretical improvements. Such an implementation would serve as a testbed for new fundamental improvements to CALPHAD modeling. A natural place to start would be to contribute modifications to an existing CALPHAD software package. This unfortunately turns out not to be a feasible path. Because the source needs to be publicly available for modifications to be possible, we exclude the multitude of closed-source commercial packages available [2–4]. The availability of open-source CALPHAD packages is limited. The most notable open-source CALPHAD package is OpenCalphad [5], which is architecturally very similar to the commercial Thermo-Calc package. The quality of the implementation is high but the architecture of the system does not allow the kind of direct manipulation of phase models at run-time necessary for CALPHAD model prototyping and database development.

A more promising architecture using the Python programming language was developed in the Gibbs package [6]. In particular the symbolic construction (as opposed to "hard-coding") of phase models makes manipulation vastly simpler. However their implementation is much more limited; Gibbs lacks support for the compound energy formalism (CEF) [7]. Moreover, Gibbs' equilibrium calculation method is based on Quickhull [8], and Quickhull-based solvers will not perform well for multicomponent systems due to the

poor scaling of general convex hull algorithms in high dimensions. (For energy minimization it is only necessary to compute the lower convex hull.) Because of these issues the majority of the core would have to be completely rewritten for our purposes. Development by the Gibbs team appears to have ceased after the original publication, so such a significant undertaking would have to be performed alone.

What is desired is something which takes the rigorous theoretical approach of Open-Calphad and combines it with the extensibility and modularity of the Python-based approach pioneered by the Gibbs package. This is the purpose of the pycalphad project.

The pycalphad software package is roughly 3000 lines of Python code designed to solve the multi-component, multi-phase Gibbs energy minimization problem with full support for the CEF. The key feature of pycalphad is that the thermodynamic models of individual phases and their associated databases can be programmatically manipulated and overridden at run-time, without modifying any internal solver or calculation code: the representation of the models is decoupled from the equilibrium solver, and the models themselves are represented symbolically. This makes pycalphad ideal for prototyping CALPHAD models and developing CALPHAD databases.

## Implementation and architecture

**Figure 1** depicts the general architecture of pycalphad. Model parameters and associated inputs are represented as a `Database` object. Using parameters from the given `Database`, a `Model` object is constructed for each phase containing the symbolic representation of that phase's energy function. These symbolic representations are then fed into the calculation engine to produce results for the user.

**Figure 2** illustrates the benefit of this modular architecture for creating custom phase models. Creating a custom model in pycalphad involves creating a subclass of the `Model` class. The key step is declaring the `contributions` class attribute.

The attribute is a list of tuples, and each tuple contains a unique name and a class function name. The class function is called to construct the corresponding energetic contribution, with several contributions already defined in `Model`. In this case, the custom energetic contribution is the system temperature multiplied by the squared deviation from the equimolar composition of the phase. The `CustomModel` subclass can then be passed as an argument to `calculate()` and `equilibrium()`.

### *Database*

The `Database` object is the fundamental representation of CALPHAD data in pycalphad. It can be considered as the in-memory analogue to a thermodynamic database (TDB) file, and in fact pycalphad supports reading and writing a large subset of the TDB file format through the `from_file()` and `to_file()` methods of the `Database` object. For convenience, calling the `Database` constructor, e.g., `Database('example.tdb')`, will automatically call the appropriate parsing function. Database files can also be passed as multi-line strings; this is convenient for embedding TDB files in short Python scripts. The current version of pycalphad (0.4.2) only supports TDB files, but new file formats could be easily implemented without modifying any other code dependent on a `Database` since the objects are not coupled to any particular file format. Using this scheme it is unnecessary for the rest of pycalphad to know anything about how CALPHAD data is represented on disk.

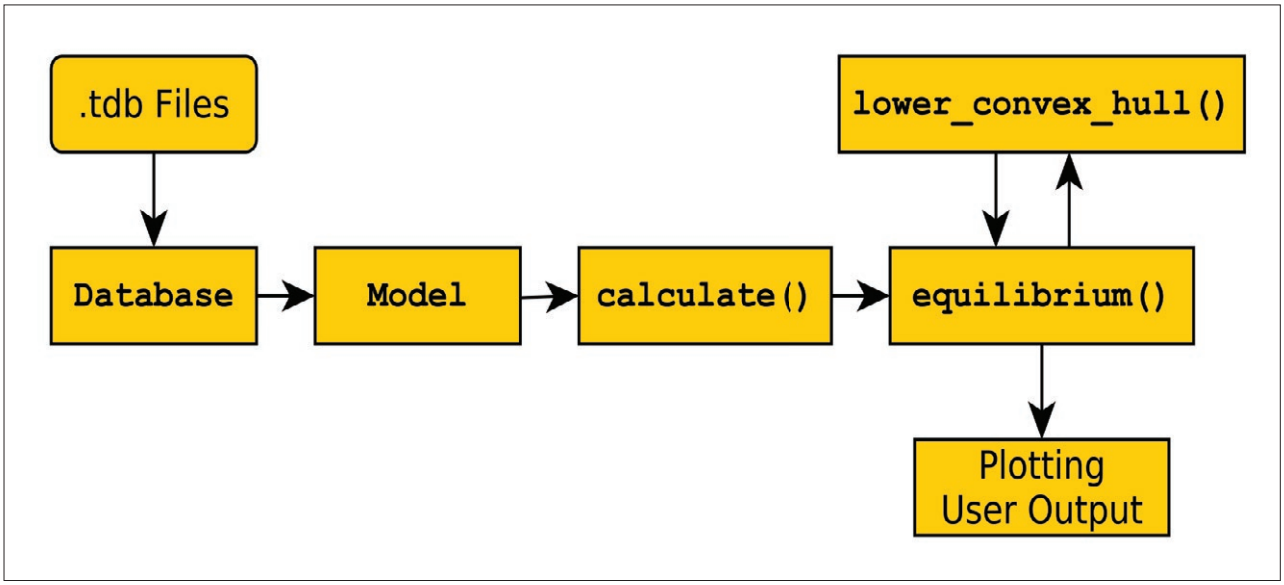


**Figure 1:** General architecture of the pycalphad software package. Using parameters from the given `Database`, a `Model` object is constructed for each phase and then fed into the calculation engine to produce results for the user.

```
from pycalphad import Model
import pycalphad.variables as v
import sympy

class CustomModel(Model):
    contributions = [('ref', 'reference_energy'),
                     ('idmix', 'ideal_mixing_energy'),
                     ('xsmix', 'excess_mixing_energy'),
                     ('mag', 'magnetic_energy'),
                     ('custom', 'custom_energy'),
                     ('ord', 'atomic_ordering_energy')]
    def custom_energy(self, phase, param_search):
        result = sympy.S.Zero
        for component in self.components:
            result += v.T * (self.standard_mole_fraction(component) - \
                1/len(self.components)) ** 2
        return result / self._site_ratio_normalization(phase)
```

**Figure 2:** Creating a custom model in pycalphad involves creating a subclass of the `Model` class. The key step is declaring the `contributions` class attribute. The `CustomModel` subclass can then be passed as an argument to `calculate()` and `equilibrium()`.

`Database` exposes a `phases` attribute containing a Python dictionary mapping the name of a phase to an object which contains information about the sublattice model and phase constituents. It also exposes a `search()` function for finding model parameters satisfying certain criteria. These are the primary features used by the `Model` object to build the symbolic representation of a phase's thermodynamic model.

*Model*

A `Model` is an abstract representation of the molar Gibbs energy function of a phase. This representation is built around the computer algebra library SymPy [9], allowing the variables and arithmetic functions required by the CEF to be represented as an abstract graph of Python objects. For example, the operation $2 + 3x$ might internally be represented as `Add(2,Mul(3,x))`, with larger structures for more complicated models. For convenience, the library `Model` class defines several thermodynamic properties. By default, attributes are defined for the following, with Thermo-Calc-style abbreviations listed in parentheses (either are allowed): `energy (GM)`, `entropy (SM)`, `enthalpy (SM)`, `heat capacity (CPM)`, `mixing energy (MIX GM)`, `mixing entropy (MIX SM)`, `mixing enthalpy (MIX HM)`, `mixing heat capacity (MIX CPM)`, `curie temperature (TC)`, and `degree of ordering (DOO)`. It is also possible for users to define custom properties for particular purposes. These SymPy-based abstract graphs, as well as their exact first and second derivatives, are compiled to machine code on demand for computational efficiency. SymPy's automatic code generation feature is used to provide users maximum flexibility since it offers the ease-of-use of working in Python without having to make a significant performance tradeoff, compared to working only in C, Fortran, or another low-level programming language.

By default the library `Model` class is used for all phases. It includes support for multi-component Redlich-Kister polynomials using the Muggianu ternary extension [10], the Inden-Hillert-Jarl magnetic model [11, 12], and the order-disorder model for atomic ordering [13, 14]

For parametric model contributions, users can use the `param_search` argument defined in the function signature of every energetic contribution to query a `Database` for parameters satisfying some criteria. The `Model` class defines a `redlich_kister_sum()` convenience function to allow users to easily build multi-component Redlich-Kister polynomials using parameters defined in a `Database`. For example, to construct the symbolic form of the mean magnetic moment of a phase in Redlich-Kister form, inside `custom_energy()` one could write

```
from tinydb import where
bm_param_query = (
    (where('phase_name') == phase.name) & \
    (where('parameter_type') == 'BMAGN') & \
    (where('constituent_array').
     test(self._array_validity))
)
mean_magnetic_moment = \
    self.redlich_kister_sum(phase, param_search,
    bm_param_query)
```

This code snippet will pull all the relevant magnetic parameters from the database, filtered by `self._array_validity` to include only the declared components in our model.

*calculate()*

The `calculate()` function is the core property calculation routine of pycalphad. It does not concern itself with equilibrium at all – that is the responsibility of `equilibrium()` – but instead performs calculations for the case when all independent degrees of freedom, i.e., temperature, pressure, sublattice site fractions, are specified. The most important arguments of `calculate()` are

```
def calculate(dbf, comps, phases, output='GM',
              model=None, points=None,
              T=None, P=None, **kwargs)
```

`dbf` is the `Database` containing the relevant parameters, `comps` is a list of desired components for the calculation, and `phases` is a list of desired phases. (Users can get a list of all phases using `list(dbf.phases.keys())`.) By default `calculate()` will compute the `GM` property of all phases, but users can specify any property defined by the phase model, including properties defined in custom models. By default the library `Model` class is used for all phases.

Custom models can be specified via the `model` keyword argument. For example, `calculate(dbf, comps, phases, model=CustomModel)` overrides the default model for all phases in that energy calculation. To override only a specific phase's model, we write `model={'FCC_A1': CustomModel}` to override the model for the `FCC_A1` phase. More sophisticated formulations are also possible. We can use `model=[{'FCC_A1': CustomModel, 'LIQUID': Model}, YetAnotherModel]` for the `FCC_A1` phase to use `CustomModel`, the liquid phase to use the library `Model`, and all other phases in the calculation to use `YetAnotherModel` (not defined here). The `output` keyword argument specifies the property to calculate; this is a string corresponding to an attribute of the library `Model` or a user-defined subclass of `Model`, as discussed above. For example, we write `output='CPM'` to indicate the molar heat capacity should be computed. If `output` is not specified, by default only the molar Gibbs energy is calculated.

The `points` keyword argument accepts a multi-dimensional array of shape (*P, T, y*), where *P* and *T* are pressures and temperatures at which to perform the calculation, and `y` is the number of sublattice site fractions. Site fractions are ordered by sublattice number, then alphabetically within a sublattice, e.g., $y^0_{Al}, y^0_{Ni}, y^0_{Cr}, y^0_{Mo}, y^1_{Ni}, y^1_{Nb}$. If the same site fractions are meant to be used for all temperatures and pressures in the calculation, the *P* and *T* dimensions can be omitted from the array. For multi-phase calculations, users can pass a Python dictionary mapping the name of a phase to an array of site fractions.

The *T* and *P* keyword arguments are the temperatures and pressures in Kelvin and pascals, respectively, for the calculation. (Specifically the units are whatever *T* and *P* mean in the phase model but, in the default `Model`, SI units are used.) Valid arguments are either a scalar or a one-dimensional array. `**kwargs` is a placeholder for other, less commonly used or experimental options; these are discussed in the pycalphad documentation linked from the GitHub repository. The return value of `calculate()` is a multi-dimensional labeled array.

*equilibrium()*

The `equilibrium()` function is responsible for equilibrium property calculation in pycalphad. Its key arguments are

```
def equilibrium(dbf, comps, phases, conditions,
                output=None, model=None, **kwargs)
```

`dbf`, `comps`, `phases`, `output`, `model`, and `**kwargs` all have the same meaning as in the `calculate()` function, with the additional feature that `output` can be either a string or a list of strings. `conditions` is a Python dictionary mapping state variables to values. Valid arguments for a condition are a scalar, one-dimensional array, or tuple with the form (*start, stop, step*). For example, an isothermal step calculation might have a `conditions` argument of the form `{v.X('AL'): (0,1, 0.01), v.T: 600}`, where `v` is defined as a shortcut to `pycalphad.variables`, a library module where all standard symbols are defined.

The return value of `equilibrium()` is a multi-dimensional labeled array. Regardless of the value of `output`, the result array will always include the equilibrium values of the molar Gibbs energy and chemical potentials since they are necessary to compute the solution.

### Representation of results

The result of calls to `calculate()` and `equilibrium()` are xarray Dataset objects [15]. The xarray Dataset object makes handling labeled multi-dimensional arrays substantially simpler. **Figure 3** shows the xarray summary of the result of a 2-D mapping calculation. The "Dimensions" line indicates the shape of the array, with each dimension having a label and corresponding size. In this case, equilibria at 170 temperatures and 100 compositions are computed for a two component system.

The "internal dof" dimension corresponds to the sublattice site fractions of a phase. For phases with fewer than the maximum number of internal degrees of freedom, the extra elements are filled with NaN (Not A Number). The "vertex" dimension corresponds to the vertices of a tie simplex (tie-line in binary systems). For single- phase regions, only the first vertex is valid and the others are filled with NaN.

The "Data Variables" section contains the actual result of the calculation, with the corresponding dimensions of each property array listed in parentheses, followed by the first few values. The "Phase" and "NP" arrays contain the names and fractions of

```
Dimensions:             (P: 1, T: 170, X_AL: 100, component: 2, internal_dof:
 ↪  5, vertex: 2)
Coordinates:
  * P                   (P) float64 1.013e+05
  * T                   (T) float64 300.0 310.0 320.0 330.0 340.0 350.0 ...
  * X_AL                (X_AL) float64 1e-09 0.01 0.02 0.03 0.04 0.05 0.06 ...
  * vertex              (vertex) int64 0 1
  * component           (component) object 'AL' 'FE'
  * internal_dof        (internal_dof) int64 0 1 2 3 4
Data variables:
    MU                  (P, T, X_AL, component) float64 -1.569e+05 ...
    NP                  (P, T, X_AL, vertex) float64 1.0 nan 1.0 nan 1.0 nan
     ↪  ...
    GM                  (P, T, X_AL) float64 -8.184e+03 -9.283e+03 ...
    X                   (P, T, X_AL, vertex, component) float64 1e-09 1.0 ...
    Y                   (P, T, X_AL, vertex, internal_dof) float64 1e-09 1.0
     ↪  ...
    Phase               (P, T, X_AL, vertex) object 'B2_BCC' '' 'B2_BCC' '' ...
    curie_temperature   (P, T, X_AL) float64 1.043e+03 1.028e+03 1.013e+03 ...
    degree_of_ordering  (P, T, X_AL, vertex) float64 7.397e-10 nan 3.331e-15
     ↪  ...
    heat_capacity       (P, T, X_AL) float64 24.89 24.91 24.92 24.94 24.95 ...
Attributes:
    hull_iterations: 1
    solve_iterations: 122684
    engine: pycalphad 0.3.5+11.g2f209e3.dirty
    created: 2016-05-23 17:21:26.642833
```

**Figure 3:** This is a summary of the result object returned by a call to `equilibrium()` when performing a 2-D mapping calculation. The "Dimensions" line indicates the shape of the array, with each dimension having a label and corresponding size. In this case, equilibria at 170 temperatures and 100 compositions are computed for a two component system. The "internal dof" dimension corresponds to the site fractions of a phase. The "vertex" dimension corresponds to the vertices of a tie simplex (tie-line in binary systems). The "Data Variables" section contains the actual result of the calculation, with the corresponding dimensions of each property array listed in parentheses, followed by the rst few values. The "Attributes" section contains some metadata about the calculation.

phases, respectively, present under the corresponding conditions. All the properties we specify in the `output` keyword argument are included here. The xarray library makes selecting and slicing the data very easy; for example, to get all the Al chemical potentials at 600 K, we write `eq.MU.sel(component='AL', T=600)`, where `eq` is the result array. Note that a current limitation is that the selection must correspond directly to a calculated value; automatic interpolation of the pressure, temperature, or composition is not currently implemented.

The "Attributes" section contains some metadata about the calculation such as the version of pycalphad used, the calculation date, and the solver iterations.

Datasets have functions for reading from and writing to disk, making storage of the results of long-running calculations easier. Interested users are encouraged to review the xarray documentation [15].

### *Quality control*

Even as a relatively small project, pycalphad is sufficiently complex that it is necessary to implement strategies to avoid the regression, or accidental breakage, of features. The key concepts to understand when managing and developing a complex software project are source code control (SCC) and continuous integration (CI). SCC is critical to verify the integrity of the project over time when admitting changes from multiple, scattered contributors, but it is useful even in single-contributor projects because SCC systems serve as a semi-automated project journal and backup system. This project uses the popular Git SCC system [16] to manage its source code. This allows the complete history of changes to be recorded for all released and unreleased versions of the software. Git also allows different versions of the software to be stored in separate "branches," allowing concurrent work on, e.g., new major features and bug fixes to existing versions. The Git repository is publicly available online at GitHub (see section 2). Git also extends into pycalphad's versioning system: `major.minor.rev+N.gHASH`, where `HASH` is the Git commit identifier of the latest commit in the master branch of the repository, and `N` is the number of commits ahead of the last public release. For public releases, everything after the `+` is omitted. For modifications which have not yet been committed, i.e., in a developer's local Git

repository, the version identifier will be appended with `dirty`.

CI is the approach of a project to simplify the software release process by testing code incrementally, i.e., every time a revision is made. This makes releasing new versions easier because the release manager can have some confidence that the quality of the code is above some automatically verified baseline. The pycalphad package has a suite of CI tests designed to verify that a revision to the code does not cause unintended behavior. These tests are run automatically every time a new revision is pushed to the Git repository on GitHub. If a test fails for any reason, a report is generated including all the error information. For example, there are tests to ensure that computed values of properties for several known systems do not change. The equilibrium solver, TDB reading and writing, and phase model construction code are also tested for consistency and accuracy. When a bug is reported and fixed in pycalphad, a minimal test case is added to the suite whenever possible to prevent the problem from appearing again in a future release. In total, about 80% of pycalphad, measured by lines of code, is currently tested, with the remainder involving unreachable or experimental code, or code which is difficult to test in an automated fashion, e.g., plotting code.

## (2) Availability

### Operating system

A version of Linux, OSX, or Windows capable of running a supported version of Python is required.

### Programming language

Python 2.7+ or Python 3.4+ is required.

### Additional system requirements

At least 2 GB of RAM is recommended.

### Dependencies

- gcc, MinGW or Microsoft Visual C++ compiler and toolchain
- matplotlib [17]
- numpy ≥ 1.9 [18]
- scipy [18]
- sympy [19]
- xarray [15]
- pyparsing [20]
- tinydb

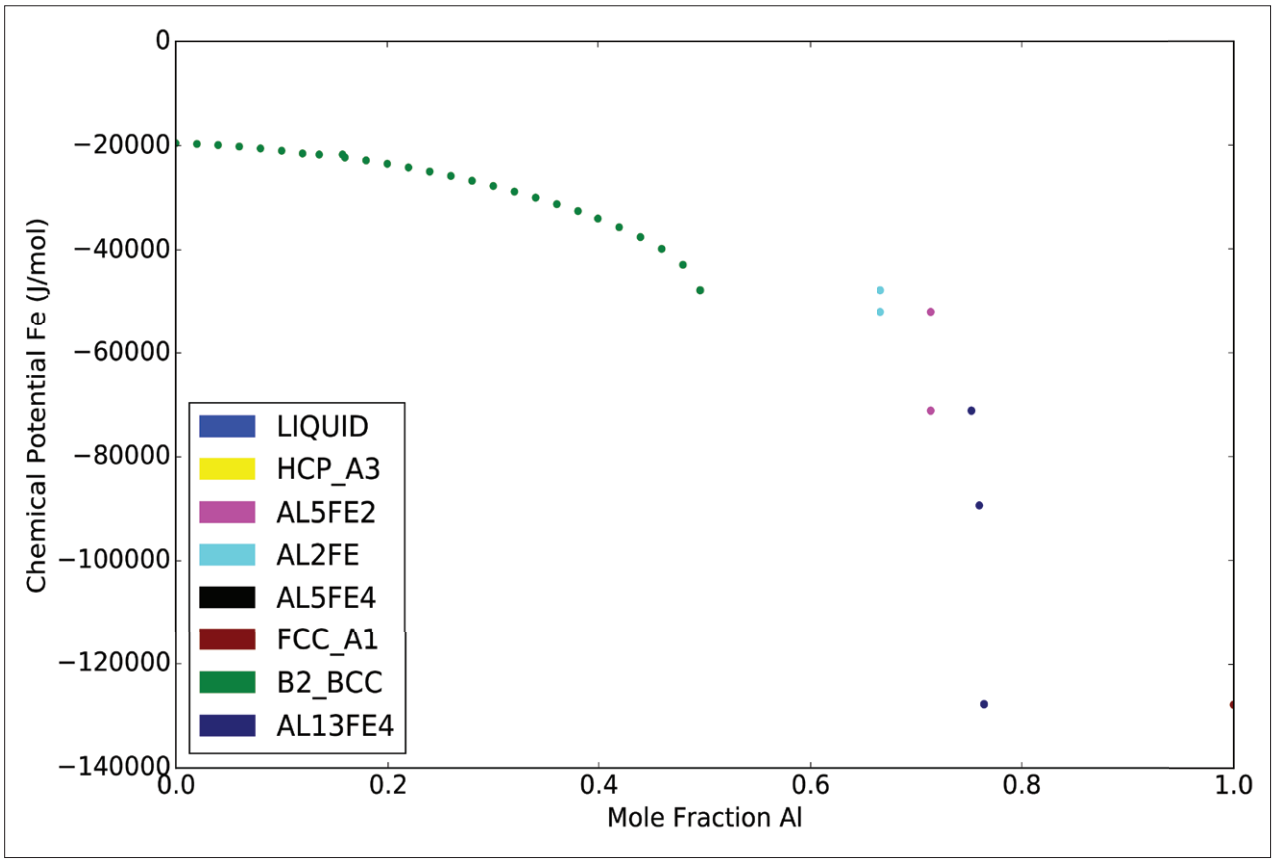


**Figure 4:** Equilibrium chemical potential of Fe as a function of Al composition in the Al-Fe system at 600 K, computed using pycalphad. Each point is color-coded with the corresponding stable phase; coexistence regions can be identied by the chemical potential remaining at across a range of composition. The end-points of such an iso-potential region can be directly connected to the corresponding tie-line at the given temperature.

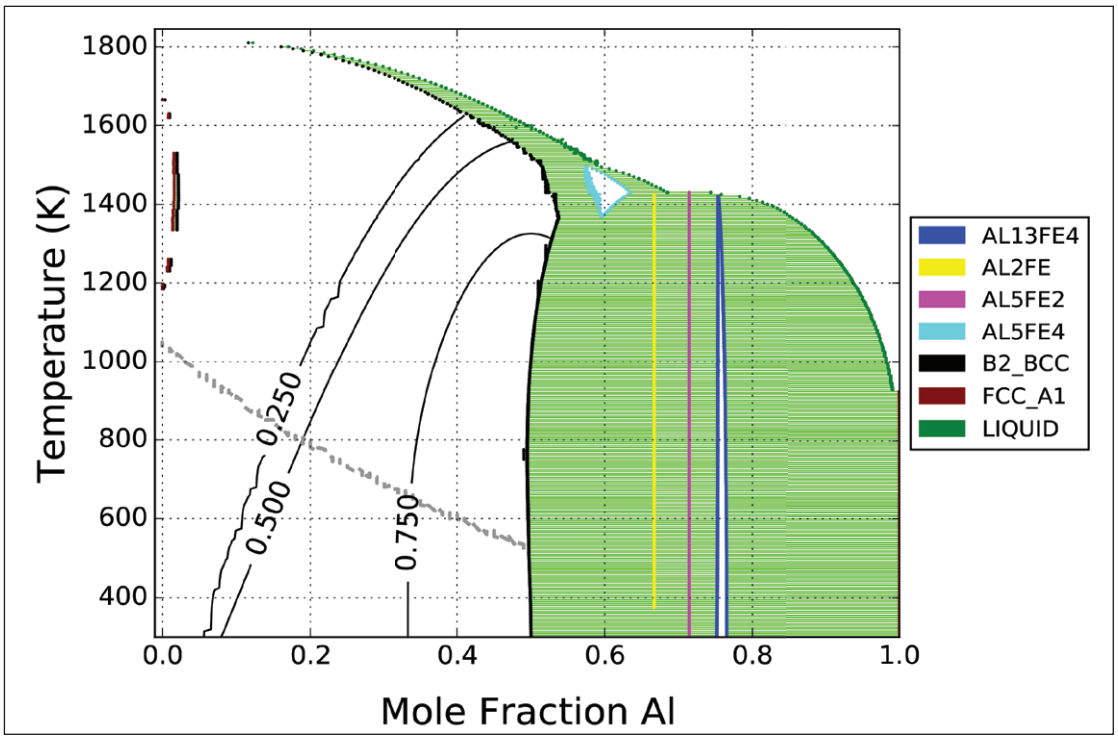


**Figure 5:** Phase diagram of the Al-Fe system according to the COST 507 database, computed using pycalphad. The solid black lines in the B2 region correspond to lines of constant "degree of ordering" in the B2 phase. The grey dashed line is the Curie temperature. The bcc ordering transition is second-order since the degree of ordering is continuously changing with respect to composition and temperature. Some lines in the diagram are not smooth due to the coarseness of the grid used in the calculation; mapping in pycalphad is still experimental.

- autograd
- tqdm
- dask
- dill

### List of contributors

Richard Otis (Pennsylvania State University) – Development and testing

Zi-Kui Liu (Pennsylvania State University) – Project supervision

### Software location

***Archive***

**Name:** Figshare

**Persistent identifier:** https://dx.doi.org/10.6084/m9.figshare.4213689

**Licence:** MIT

**Publisher:** Richard Otis

**Version published:** 0.4.2

**Date published:** 07/11/16

### Code repository

**Name:** GitHub

**Persistent identifier:** https://github.com/pycalphad/pycalphad

**Licence:** MIT

**Date published:** 09/11/16

### Language

English

## (3) Reuse potential

Al-Fe is chosen as an example system because the COST 507 database [21] containing this subsystem is publicly available, and it allows us to test several pycalphad features simultaneously since the system contains single-sublattice solution phases, multi-sublattice ordered phases, phases with magnetic ordering and stoichiometric compounds. The pycalphad package can perform computations for any number of components; we restrict our example to a binary system only for the simplicity of visualization.

**Figure 4** shows the result of a one-dimensional ("step") equilibrium calculation at 600 K. The equilibrium chemical potential of Fe is shown as a function of Al composition. Each point is color-coded with the corresponding stable phase; coexistence regions can be identified by the chemical potential remaining flat across a range of composition. The end-points of such an iso-potential region can be directly connected to the corresponding tie-line at the given temperature. The source code for this calculation can be found in **Figure 6**.

**Figure 5** shows the phase diagram of the Al-Fe system according to the COST 507 database. The solid black lines in the B2 region correspond to lines of constant "degree

```
from pycalphad import equilibrium
from pycalphad import Database, Model
import pycalphad.variables as v
import matplotlib.pyplot as plt
from pycalphad.plot.utils import phase_legend

# https://github.com/pycalphad/pycalphad/blob/master/examples/alfe_sei.TDB
db_alfe = Database('alfe_sei.TDB')
my_phases_alfe = ['LIQUID', 'HCP_A3', 'AL5FE2', 'AL2FE',
                  'AL5FE4', 'FCC_A1', 'B2_BCC', 'AL13FE4']
temp = 600
eq = equilibrium(db_alfe, ['AL', 'FE', 'VA'], my_phases_alfe,
                 {v.X('AL'): (0,1,0.02), v.T: temp, v.P: 101325})

fig = plt.figure(figsize=(14,9))
fig.gca().set_xlim((0,1))
fig.gca().set_xlabel('Mole Fraction Al', fontsize=18)
fig.gca().set_ylabel('Chemical Potential Fe (J/mol)', fontsize=18)
fig.gca().tick_params(axis='both', which='major', labelsize=18)

phase_handles, phasemap = phase_legend(my_phases_alfe)
phasecolors = [phasemap[str(p)] for p in \
               eq.Phase.sel(T=temp, vertex=0).values[0] if p != '']
fig.gca().scatter(eq.X.sel(T=temp, component='AL', vertex=0),
                  eq.MU.sel(T=temp, component='FE'), color=phasecolors)
phasecolors = [phasemap[str(p)] for p in \
               eq.Phase.sel(T=temp, vertex=1).values[0] if p != '']
fig.gca().scatter(eq.X.sel(T=temp, component='AL', vertex=1),
                  eq.MU.sel(T=temp, component='FE'), color=phasecolors)
fig.gca().legend(phase_handles, my_phases_alfe,
                 loc='lower left', fontsize=20)
fig.savefig('alfe-chempot.pdf')
```

**Figure 6:** Source code for the Al-Fe chemical potential calculation in Figure 4.

of ordering" in the B2 phase. The grey dashed line is the Curie temperature. It is clear from the diagram that the bcc ordering transition is second-order since the degree of ordering is continuously changing with respect to composition and temperature. Some lines in the diagram are not smooth due to the coarseness of the grid used in the calculation; mapping in pycalphad is still experimental. The source code for this calculation can be found in **Figure 7**.

Those interested in collaborating on pycalphad or seeking support should contact the present authors via e-mail or visit the official project website at pycalphad.org, where additional documentation and examples can be found.

**Acknowledgements**
Stimulating discussions with Bo Sundman and Neal Kelly are highly appreciated.

**Competing Interests**
The authors have no competing interests to declare.

```
import matplotlib.pyplot as plt
import pycalphad.core.constants
# Hack to fix depiction of B2 on the phase diagram
pycalphad.core.constants.COMP_DIFFERENCE_TOL = 0.1
from pycalphad import equilibrium
from pycalphad import Database, Model
from pycalphad import eqplot
import pycalphad.variables as v
import numpy as np
# https://github.com/pycalphad/pycalphad/blob/master/examples/alfe_sei.TDB
db = Database('alfe_sei.TDB')
my_phases = list(set(db.phases.keys()) - {'BCC_A2'})
eq = equilibrium(db, ['AL', 'FE', 'VA'], my_phases,
                 {v.X('AL'): (0, 1, 0.01),
                  v.T: (300, 1850, 5), v.P: 101325},
                 output=['heat_capacity', 'degree_of_ordering',
                         'curie_temperature'],
                 calc_opts={'pdens': 500})
print(eq)
fig = plt.figure(figsize=(9,6))
# Plot the phase diagram
eqplot(eq, ax=fig.gca())
# Add the ordering contours and magnetic transition line
tc_indices = np.logical_and(np.abs(eq['curie_temperature'].values - \
                            eq['T'].values[..., None]) < 10,
                            np.any(eq.Phase.values == 'B2_BCC', axis=-1))
tc_indices = np.nonzero(np.logical_and(tc_indices,
                                       np.sum(eq.Phase.values != '',
                                              axis=-1, dtype=np.int) == 1))
bcc_indices = np.logical_and(np.any(eq.Phase.values == 'B2_BCC',
                                    axis=-1),
                             np.sum(eq.Phase.values != '',
                                    axis=-1, dtype=np.int) == 1)
tc_arr = np.array([eq['X'].sel(component='AL', vertex=0).values[tc_indices],
                   np.take(eq['T'].values, tc_indices[1])])
tc_arr = tc_arr[:, tc_arr[0].argsort()]
fig.gca().plot(tc_arr[0], tc_arr[1], '--', color='grey', linewidth=2)
X, Y = np.meshgrid(eq['X_AL'].values, eq['T'].values)
doo_arr = np.ma.array(eq['degree_of_ordering'].sel(vertex=0).values,
                      mask=~bcc_indices)
CS = fig.gca().contour(X, Y, np.squeeze(doo_arr), 3, colors='k')
fig.gca().clabel(CS, inline=1, fontsize=15)
fig.gca().set_title('')
fig.gca().set_xlabel('Mole Fraction Al')
fig.savefig('alfe-with-ordering.pdf')
```

**Figure 7:** Source code for the Al-Fe phase diagram in Figure 5.

## References


1. **Spencer, P** 2008 A brief history of CALPHAD, *Calphad* 32(1): 1–8. URL http://www.sciencedirect.com/science/article/pii/ S0364591607000764. DOI: https://dx.doi.org/10.1016/j calphad.2007.10.001
2. **Cao, W, Chen, S-L, Zhang, F, Wu, K, Yang, Y, Chang, Y, Schmid-Fetzer, R** and **Oates, W** 2009 Oates PANDAT software with PanEngine, PanOptimizer and Pan-Precipitation for multi-component phase diagram calculation and materials property simulation, *Calphad* 33 (2): 328–342. URL http://www.sciencedirect.com/science/article/pii/S0364591608000709. DOI: https://dx.doi.org/10.1016/j.calphad.2008.08.004
3. **Andersson, J-O, Helander, T, Höglund, L, Shi, P,** and **Sundman, B** 2002 Thermo-Calc & DICTRA, computational tools for materials science, *Calphad* 26 (2): 273–312. DOI: https://dx.doi.org/10.1016/S0364-5916(02)00037-8
4. **Bale, C, Bélisle, E, Chartrand, P, Decterov, S, Eriksson, G, Hack, K, Jung, I-H, Kang, Y-B, Melançon, J, Pelton, A, Robelin, C,** and **Petersen, S** 2009 FactSage thermochemical software and databases recent developments, *Calphad* 33 (2): 295–311. DOI: http://dx.doi.org/10.1016/j.calphad.2008.09.009
5. **Sundman, B, Kattner, U R, Palumbo, M,** and **Fries, S G** 2015 OpenCalphad – a free thermodynamic software, *Integr. Mater. Manuf. Innov.* 4 (1): 1. URL http://www.immijournal.com/content/4/1/1. DOI: https://dx.doi.org/10.1186/s40192-014-0029-1
6. **Cool, T, Bartol, A, Kasenga, M, Modi, K,** and **García, R E** 2010 Gibbs: Phase equilibria and symbolic computation of thermodynamic properties, *Calphad* 34 (4): 393–404. URL http://www.sciencedirect.com/science/article/pii/ S0364591610000507. DOI: https://dx.doi.org/10.1016/j.calphad.2010.07.005
7. **Hillert, M** 2001 The compound energy formalism, *J. Alloys Compd.* 320 (2): 161–176. DOI: https://dx.doi.org/10.1016/S0925-8388(00)01481-X
8. **Barber, C B, Dobkin, D P,** and **Huhdanpaa, H** 1996 The quickhull algorithm for convex hulls, *ACM Trans. Math. Softw.* 22 (4): 469–483. DOI: https://dx.doi.org/10.1145/235815.235821
9. **Joyner, D, Čertík, O, Meurer, A,** and **Granger, B E** 2012 Open source computer algebra systems, *ACM Commun. Comput. Algebr.* 45 (3/4): 225. DOI: https://dx.doi.org/10.1145/2110170.2110185
10. **Muggianu, Y M, Gambino, M,** and **Bros, J P** 1975 Enthalpies of formation of liquid alloys bismuth-gallium-tin at 723k-choice of an analytical representation of integral and partial thermodynamic functions of mixing for this ternary-system, *J. Chim. Phys. Physico-Chimie Biol.* 72 (1): 83–88.
11. **Inden, G** 1976 Approximate description of the configurational specific heat during a magnetic order-disorder transformation, in: Proc. CALPHAD V, Max Planck Institut fuer Eisenforschung, Dusseldorf, Germany, pp. 1–13.
12. **Hillert, M,** and **Jarl, M** 1978 A model for alloying in ferromagnetic metals, *Calphad* 2(3): 227–238. DOI: https://dx.doi.org/10.1016/0364-5916(78)90011-1
13. **Ansara, I, Sundman, B,** and **Willemin, P** 1988 Thermodynamic modeling of or- dered phases in the Ni-Al system, *Acta Metall.* 36 (4): 977–982. DOI: https://dx.doi.org/10.1016/0001-6160(88)90152-6
14. **Ansara, I, Dupin, N,** and **Sundman, B** 1997 Reply to the paper: When is a compound energy not a compound energy? A critique of the 2-sublattice order/disorder model, *Calphad* 21 (4): 535–542. DOI: https://dx.doi.org/10.1016/S0364-5916(98)00010-8
15. **Hoyer, S, Kleeman, A,** and **Brevdo, E** 2016 N-D labeled arrays and datasets in Python xarray 0.7.2 documentation. URL http://xarray.pydata.org/en/stable/.
16. **Chacon, S** and **Straub, B** 2014 Pro Git, Apress. DOI: https://dx.doi.org/10.1007/978-1-4842-0076-6
17. **Hunter, J D** 2007 Matplotlib: A 2D graphics environment, *Comput. Sci. Eng.* 9 (3): 90–95. DOI: https://dx.doi.org/10.1109/MCSE.2007.55
18. **van der Walt, S, Colbert, S C,** and **Varoquaux, G** 2011 The NumPy Array: A Structure for Efficient Numerical Computation. *Comput. Sci. Eng.* 13 (2): 22–30. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5725236. DOI: https://dx.doi.org/10.1109/MCSE.2011.37
19. **Meurer, A, Smith, C P, Paprocki, M, Čertík, O, Kirpichev, S B, Rocklin, M, Kumar, A, Ivanov, S, Moore, J K, Singh, S, Rathnayake, T, Vig, S, Granger, B E, Muller, R P, Bonazzi, F, Gupta, H, Vats, S, Johansson, F, Pedregosa, F, Curry, M J, Terrel, A R, Roučka, Š, Saboo, A, Fernando, I, Kulal, S, Cimrman, R,** and **Scopatz, A** SymPy: Symbolic computing in Python. DOI: https://dx.doi.org/10.7287/PEERJ.PREPRINTS.2083V3
20. **McGuire, P** 2007 Getting started with pyparsing, O'Reilly.
21. **Ansara, I, Dinsdale, A,** and **Rand, M** 1998 COST 507 Thermochemical database for light metal alloys (jul). URL https://materialsdata.nist.gov/dspace/xmlui/handle/11256/618.

**How to cite this article:** Otis, R and Liu, Z-K 2017 pycalphad: CALPHAD-based Computational Thermodynamics in Python. *Journal of Open Research Software,* 5: 1, DOI: http://dx.doi.org/10.5334/jors.140

**Submitted:** 09 August 2016 **Accepted:** 10 November 2016 **Published:** 09 January 2017



]u[ *Journal of Open Research Software* is a peer-reviewed open access journal published by Ubiquity Press

OPEN ACCESS